AD-A037 638 IBM FEDERAL SYSTEMS DIV GAITHERSBURG MD F/G 9/2

CANDIDATE LANGUAGE EVALUATION AND RECOMMENDATION REPORT.(U)

1977 F30602-77-C-0009

UNCLASSIFIED NL

1 OF 2

ADA037638

1

5.3.4

6

Candidate Language Evaluation and Recommendation Report

12 144 p.

11 1977

15

Performed Under Contract No. F30602-77-C-0009

for

ROME AIR DEVELOPMENT CENTER
GRIFISS AIR FORCE BASE
Rome, New York

International Business Machine ~~Corporation~~
Federal Systems Division
18100 Frederick Pike
Gaithersburg, Maryland 20760

174 950

mt

ABSTRACT

This report contains comparisons between three existing High-Order Languages (HOL) - COBOL, FORTRAN and HAL/S - and the DoD identified HOL requirements for embedded computers as published in the "Requirements for High-Order Computer Programming Languages 'TINMAN'" June 1976. FORTRAN and COBOL are standard DoD High-Order Programming Languages, which address scientific and business programming. HAL/S is a NASA-designed HOL for use on the On-Board Shuttle and other computers. The study was performed for Rome Air Development Center, in support of DoD's HOL cost reduction project through HOL standardization.

The results of this study show that FORTRAN meets relatively few of the requirements for a HOL for embedded computers. COBOL meets a much larger number of the TINMAN requirements and is particularly readable (and hence maintainable) because of its resemblance to English. However, COBOL is not well thought of by the computer-science community because of its wordiness and the inelegance of its program structure. Both FORTRAN and COBOL are widely used languages both by industry and government for non-embedded computer applications, and have been implemented on most general-purpose computers.

HAL/S was designed for programming embedded computers and satisfies an even larger number of the TINMAN requirements than COBOL. The limited number of implementations speaks both for and against HAL/S: it is not widely known and used, but it has a very high degree of standardization.

ACKNOWLEDGEMENTS

This report was produced in response to Task 4.1.3 in the Statement of Work for the High-Order Language (HOL) effort under Contract Number F30602-77-C-0003. It was delivered to RADC in accordance with Item A002 of the Contract Data Requirements List.

This report was prepared by:

Jean E. Sammet
Maurice Ackroyd
Michael L. Bell
I. Gray Kinnie
Richard S. Kopp

Significant contributions were made by:

Allen R. Mendelin
Thomas Spillman

HAL/S compiler advice was provided by:

Daniel J. Lickly of Intermetrics, Inc.

Contributions were made by the following RADC personnel:

Clement D. Falzarano
Douglas A. White
Samuel A. DiNitto

and the following HOL Working Group personnel:

LTC. William Whitaker of DARPA
Dr. Serafino Amoroso of ECOM
Dr. David Fisher of IDA
Dr. Peter Wegner of Brown University

TABLE OF CONTENTS

Section I - EXECUTIVE SUMMARY

Section Ia - Executive Summary Introduction

The results described in this report were performed under Rome Air Development Center Contract Number F30602-77-C-0009. The effort consisted of investigation and comparison of three existing High-Order Languages (HOL) - COBOL, FORTRAN and HAL/S - to the DoD identified HOL requirements as published in "Requirements for High-Order Computer Programming Languages 'TINMAN'" June 1976. FORTRAN and COBOL are current standard DoD HOL languages. HAL/S is a NASA designed HOL for use in On-Board Shuttle computers. Based upon the comparison IBM-Federal Systems Division has indicated the degree of compliance between the requirements and each of the three languages. Included is an explanation for the compliance ratings, possible changes to bring compliance, and possible eliminations from the three languages. This report has been placed on the ARPANET for use by the DoD HOL Working Group.

The remainder of this executive summary gives a comparison evaluation matrix and three summary recommendations for the languages compared.

Section Ib - Evaluation Matrix

A summary of how well each language meets each requirement is presented below. "T" indicates the requirement is almost completely satisified, "P" indicates it is partially satisfied, "F" indicates it fails to satisfy, and "U" (for unknown) indicates it was not possible to determine whether the requirement was satisfied.

REQUIREMENT			FORTRAN	COBOL	HAL/S
A.	Data and Types				
	A1.	Typed Language	P	T	T
	A2.	Data Types	P	P	P
	A3.	Precision	F	P	T
	A4.	Fixed-Point Numbers	F	T	F
	A5.	Character Data	F	T	F
	A6.	Arrays	P	T	T
	A7.	Records	F	T	T
B.	Operations				
	B1.	Assignment and Reference	T	T	T
	B2.	Equivalence	P	T	T
	B3.	Relationals	T	T	T
	B4.	Ar'thmetic Operations	P	T	P
	B5.	Truncation and Rounding	P	T	P
	B6.	Boolean Operations	P	P	T
	B7.	Scalar Operations - On Arrays	F	T	T
	B8.	Type Conversion - Implicit	P	F	F
	B9.	Type Conversion - Explicit	P	P	P
	B10.	I/O Operations	P	T	P
	B11.	Power Set Operations	F	F	F
C.	Expressions and Parameters				
	C1.	Side Effects	T	T	T
	C2.	Operand Structure	T	T	T
	C3.	Expressions Permitted	P	F	T
	C4.	Constant Expressions	P	F	T
	C5.	Consistent Parameter Rules	T	P	T
	C6.	Type Agreement in Parameters	P	F	T
	C7.	Formal Parameter Kinds	F	F	T
	C8.	Formal Parameter Specification	F	T	F
	C9.	Variable Numbers of Parameters	F	F	F
D.	Variables, Literals and Constant				
	D1.	Constant Value Identifiers	F	T	T
	D2.	Numeric Literals	T	T	U
	D3.	Initial Values of Variables	T	T	T
	D4.	Numeric Range and Step Size	F	T	F
	D5.	Variable Types	F	T	P
	D6.	Pointer Variables	F	F	T

E.	Definition Facilities			
E1.	User Definitions Possible	P	P	P
E2.	Consistent Use of Types	F	F	F
E3.	No Default Declarations	F	F	F
E4.	Can Extend Existing Operators	F	F	F
E5.	Type Definitions	F	F	F
E6.	Data Defining Mechanisms	F	F	F
E7.	No Free Union or Subset Types	T	T	T
E8.	Type Initialization	F	F	F
F.	Scope and Libraries			
F1.	Separate Allocation and Access Allowed	F	F	T
F2.	Limiting Access Scope	F	P	T
F3.	Compile Time Scope Determination	T	T	T
F4.	Libraries Available	P	T	T
F5.	Library Contents	F	T	T
F6.	Libraries and Compools Indistinguishable	F	T	T
F7.	Standard Library Definitions	P	P	T
G.	Control Structures			
G1.	Kinds of Control Structures	P	P	P
G2.	The GOTO	T	P	T
G3.	Conditional Control	F	P	P
G4.	Iterative Control	P	T	P
G5.	Routines	F	F	F
G6.	Parallel Processing	F	F	T
G7.	Exception Handling	F	T	T
G8.	Synchronization and Real-Time	F	P	T
H.	Syntax and Comment Conventions			
H1.	General Characteristics	P	P	T
H2.	No Syntax Extensions	T	T	T
H3.	Source Character Set	T	T	T
H4.	Identifiers and Literals	P	P	T
H5.	Lexical Units and Lines	F	F	P
H6.	Key Words	F	P	T
H7.	Comment Conventions	F	P	P
H8.	Unmatched Parentheses	T	T	T
H9.	Uniform Referent Notation	P	T	T
H10.	Consistency of Meaning	F	T	F
I.	Default, Conditional Compilation and Language Restrictions			
I1.	No Defaults in Program Logic	F	T	T
I2.	Object Representation Specifications Optional	F	T	F
I3.	Compile Time Variables	F	T	T
I4.	Conditional Compilation	F	F	P
I5.	Simple Base Language	P	F	P
I6.	Translator Restrictions	P	F	F
I7.	Object Machine Restrictions	P	T	T

J.	Efficient Object Representations and Machine Dependencies			
J1.	Efficient Object Code	T	P	T
J2.	Optimizations Do Not Change Program Effect	P	U	T
J3.	Machine Language Insertions	F	P	P
J4.	Object Representation Specification	F	T	T
J5.	Open and Closed Routine Calls	F	F	P
K.	Program Environment			
K1.	Operating System Not Required	T	T	T
K2.	Program Assembly	P	T	T
K3.	Software Development Tools	T	P	T
K4.	Translator Options	U	P	T
K5.	Assertions and Other Optional Specifications	F	P	P
L.	Translators			
L1.	No Superset Implementations	F	F	T
L2.	No Subset Implementations	T	F	T
L3.	Low-Cost Translation	T	P	U
L4.	Many Object Machines	T	T	P
L5.	Self-Hosting Not Required	T	T	T
L6.	Translator Checking Required	F	U	T
L7.	Diagnostic Messages	F	F	F
L8.	Translator Internal Structure	T	T	T
L9.	Self-Implementable Language	F	T	F
M.	Language Definition, Standards and Control			
M1.	Existing Language Features Only	P	T	T
M2.	Unambiguous Definition	F	F	T
M3.	Language Documentation Required	F	P	T
M4.	Control Agent Required	F	T	T
M5.	Support Agent Required	F	U	T
M6.	Library Standards and Support Required	F	U	T
Totals	T (Fully Satisfies)	21	46	59
	P (Partially Satisfies)	27	22	18
	F (Fails to Satisfy)	49	26	19
	U (Unknown)	1	4	2

Section Ic - FORTRAN Language Executive Summary

The FORTRAN language reviewed here is "The American National Standard FORTRAN", ANSI X3.9 - 1966, known here as FORTRAN66. Reference is also made, where appropriate, to FORTRAN76: "The Draft Proposed ANSI FORTRAN" prepared by ANSI Committee X3J3 and published in ACM SIGPLAN Notices as Volume II, No. 3.

FORTRAN is not recommended as a base for the DoD HOL. It is immediately clear, on reading TINMAN, that FORTRAN cannot be modified to satisfy any significant part of the requirements, and still maintain its individuality as a language:

a. FORTRAN's traditional unstructured data specification facilities cannot survive the introduction of composite data structures (A2) particularly when fully generalized (D5). A structured data declaration such as PL/I's DCL is required; DIMENSION, INTEGER, etc., become redundant.

b. Introduction of a fully-partitioned conditional control (G3) makes all of FORTRAN's conditional and switching mechanisms redundant (Logical and Arithmetic IF; Assigned and Computed GOTO). The IF...THEN...ELSE of PL/I requires only minor change.

c. Complete type checking (C6) widens the FORTRAN "domain of compilation" from the traditional "program unit" to the complete executable program. This suggests relaxing the distinction between internally- and externally-defined procedures and leads, reasonably if not inevitably, to the elimination of the familiar FORTRAN function classes: statement functions, intrinsic functions, basic external functions, external functions (these are distinguished mainly by the degree of type checking to be performed by the translator). PL/I satisfies the requirement with only minor change.

d. User specification of allocation and access scope (F1) implies the introduction of storage classes and the elimination of COMMON. PL/I already has storage classes.

e. Reserved keywords (H6), a specific statement delimiter (H1), mnemonic statement identifiers (H1), and the required comments conventions (H7), all require changes to FORTRAN, but none to PL/I.

Thus, by considering only a few requirements, we have eliminated nearly all the distinctive features of FORTRAN - all that distinguishes FORTRAN from a rather weak subset of PL/I. By contrast, PL/I requires very little change to meet those same requirements. In fact, there is probably no positive, as distinct from restrictive, requirement that PL/I does not satisfy at least as well as FORTRAN. The conclusion is evident: PL/I is a far stronger candidate than FORTRAN, which should be eliminated from contention. It is noted that a similar case could be made for ALGOL68, and others, as against FORTRAN. PL/I has been discussed simply because it initially used FORTRAN as a base upon which to add functions to satisfy a large class of problems.

Section Id - COBOL Language Executive Summary

The language reviewed in this report is the "American National Standard Programming Language COBOL", ANSI X3.23-1974, known herein as COBOL. Reference is also made, where significant differences occur, to the "United States of American National Standard COBOL", ANSI X.23-1968, known as COBOL 68.

Although COBOL is widely used and implemented for its intended class of applications, THERE WOULD PROBABLY BE GREAT PSYCHOLOGICAL RESISTANCE TO ITS ADOPTION AS A BASIS FOR TINMAN WHATEVER COBOL'S TECHNICAL MERITS MIGHT BE WITH RESPECT TO OTHER LANGUAGES.

It is not very surprising to note that COBOL does best in TINMAN requirement areas A, B, D and F. They are, respectively: Data and Types; Operation; Variables, Literals and Constants; and Scopes and Libraries. COBOL is poorest in E (Definition Facilities), since COBOL has none aside from subroutines and does rather poorly in category C (Parameters and Expressions). The entire matter of structure of programs and parameter passage is certainly one of COBOL's weakest concepts.

COBOL 74, considered as a base language for the DoD HOL is:

a. Moderately compliant with existing TINMAN requirements

b. Easily changed to meet approximately 23 of the requirements it does not fully meet

c. Moderately easily changed in approximately 10 of the requirements it does not fully meet

d. Hard or impossible to change in approximately 13 of the requirements it does not fully meet

e. A viable base technically, but probably unacceptable psychologically because of its verbosity and past history, and because language experts have such a low opinion of it.

Note that the numbers above represent complete compliance and include some requirements rated "90%, fully satisfies" in the report.

Section Ie - HAL/S Language Executive Summary

The HAL/S Language reviewed here is described in the "HAL/S Language Specification" prepared by Neubold and Helmers of Intermetrics, Inc., for The National Aeronautics and Space Administration (NASA). The language was specifically designed for use in programming the flight software of the NASA Space Shuttle Program. HAL/S falls in the class of programming languages whose ancestry includes PL/I, but it differs considerably in features deleted and added.

Considering the total set of TINMAN requirements, HAL/S does not meet DoD specifications for a common language. However, the language could be expanded to provide some of the capabilities required, and thus make HAL/S a base for the design of a common language.

Many of the features required in the TINMAN have not been found necessary for embedded spacecraft computer systems. These include extensibility, generic procedures, and recursion. If these requirements are removed, HAL/S would only fail in ten requirements.

HAL/S, considered as a base language for the DoD HOL:

a. Is compliant with the majority of existing TINMAN requirements

b. Could be easily changed to meet approximately 13 of the requirements it does not fully meet

c. Could be changed with moderate difficulty in approximately 12 of the requirements it does not fully meet

d. Would be very difficult, but not impossible to change in approximately 14 of the remaining requirements it does not fully meet

e. Is a viable base technically, but is probably a poorer choice than some of the larger, more widely used languages.

The effort reflected in this report is in support of the DoD program for software commonality through the adoption of a minimal number of high-order programming languages.

The considerable discussion given to the establishment of a Standard Programming Language for the Department of Defense led to the creation of a Joint Services Committee on January 28, 1975. The purpose of this committee was to determine the language requirements of the services, compare existing languages to these requirements and make the necessary recommendations to DoD.

Currently the Army is using TACPOL, which was developed by Litton, based on PL/I and used to program TACFIRE and TOS. The U.S. Air Force has developed and used certain dialects of JOVIAL, based on ALGOL-58. The U.S. Navy is now using CMS-2. The three services, respectively, have developed specifications for TACPOL II, JOVIAL J73 and CS-4 as their future command and control languages. (CS-4 is based on PASCAL.)

The goal of this effort is to assist DoD in their effort toward software commonality through investigation and comparison of three identified languages to the DoD Higher Order Language (HOL) Working Group's language requirement document "Requirements for High-Order Computer Programming Languages, 'TINMAN'".

IBM-Federal Systems Division has compared the Department of Defense Requirements for High-Order Computer Programming Languages, TINMAN, individually, to ANSI FORTRAN, ANSI COBOL, and HAL/S. Each of the 98 TINMAN requirements was evaluated for degree of compliance and modifications needed. Results of these comparisons were detailed in Section Ib.

For each of the candidate languages, the evaluators determined the degree of compliance to each of the TINMAN requirements and explained how they are met. For each requirement less than fully satisfied, the conflicts were explained, as was the scope of typical modifications necessary to meet the requirement, and the impact of the changes. Other programming considerations and cross conflicts in TINMAN were addressed.

Each of the detailed requirements of TINMAN evaluated against each candidate language used the following criteria and format:

T - Fully satisfies the requirement - meets 90% or greater of the requirement

P - Partially satisfies the requirement - meets 50% to 89% of the requirement

F - Fails to satisfy the requirement - meets less than 50% of the requirement

U - Unknown from the available documents whether the requirement is satisfied or requirement is implementation-dependent (distinction will be made in the comments).

An easy-reference, the Matrix of Compliance, is provided for comparison of the three languages in Section Ib. This provides requirement number, short title of requirement, and the grades T, P, F, or U for each of the three languages.

Section I, the Executive Summary, includes the matrix and an overview of this report. Section II is this Introduction. Sections III, IV and V, respectively, describe the FORTRAN, COBOL and HAL/S evaluations. Each of these sections is organized into an introduction, detailed comparisons to TINMAN requirements to the specific language specifications, a list of unnecessary features and which can probably safely be discarded, and summary and recommendations. Section VI briefly addresses TINMAN as a HOL requirements document.

This report is organized to provide a cohesive look at how the three candidate languages (COBOL, FORTRAN and HAL/S) meet DoD's needs for a HOL for Embedded Computers (TINMAN). At the same time, the report is designd to be parsed into individual language reports (while still remaining cohesive) and eventually split into over 300 individual files on the ARPANET which address individual TINMAN requirements, paragraph by paragraph. Hence, each small group of paragraphs will have a coded heading such as "#.COBOL.Summary and Recommendations", "A. HALS.Data Types", etc.

This report does not reproduce the requirements as stated in TINMAN. Therefore, anyone desiring to read the specific details of the report should first become thoroughly familiar with TINMAN.

The primary references used in this report are:

a. "Requirements for High-Order Computer Programming Languages - 'TINMAN'", June 1976.

b. "American National Standard Programming Language COBOL," ANSI

X3.23-1974, American National Standards Institute, Inc., New York, 1974 (less the Report Writer Module).

c. "United States of American National Standard COBOL," ANSI X3.23-1968, American National Standards Institute, Inc., New York, 1968.

d. "American National Standard (ANS) FORTRAN," ANSI X3.9-1966, American National Standards Institute, Inc., New York, March 1966.

e. "Draft Proposed American National Standard FORTRAN," BSR X-3.9 X3J3/76, American National Standards Committee X3J3, March 1976.

f. Newbold, P. M. and Helmers, C. T., Jr., "HAL/S Language Specification," Intermetrics, Inc., Cambridge, Massachusetts, April 1973.

g. Augmentation Research Center, "NLS Users Guide," Stanford Research Institute, Palo Alto, California, 1976.

Supplemental references include:

a. "CODASYL COBOL Journal of Development (1976)", CODASYL Program Language Committee, Monroeville, Pennsylvania, 1976.

b. Newbold, P. M. and Intermetrics Staff, "HAL/S Programmer's Guide," Intermetrics Inc., including revisions through June 11, 1976.

Section III - FORTRAN EVALUATION

Section IIIa - FORTRAN Introduction

#.FORTRAN.Introduction

The FORTRAN language reviewed here is "The American National Standard FORTRAN", ANSI X3.9 - 1966, known here as FORTRAN66. Reference is also made, where appropriate, to FORTRAN76: "The Draft Proposed ANSI FORTRAN", prepared by ANSI Committee X3J3 and published in ACM SIGPLAN Notices as Volume II, No. 3, March 1976. References in the text to FORTRAN refer equally to both FORTRAN66 and FORTRAN76.

FORTRAN76 is a superset of FORTRAN66 (barring minor incompatibilities). Notable additions are:

a. New statements of minor importance (IMPLICIT, INTRINSIC, etc.).

b. Character string data.

c. Arrays may have seven dimensions; upper and lower subscript bounds may be specified.

d. Use of expressions is generalized throughout.

e. A significant amount of new language for I/O.

f. A considerable effort of resolution of semantic ambiguities in FORTRAN66.

Characteristics of the FORTRAN Language

Data Specification

Data specification in FORTRAN66 is unstructured. One may have in one program, for one variable, a type statement:

```
INTEGER X
```

an array declaration:

```
DIMENSION X(10,10)
```

a scope statement:

```
COMMON /C/X
```

an "association" statement:

```
EQUIVALENCE (X(1,1),J)
```

and an initialization statement:

```
DATA X(1,1)/5/
```

These declarations need not appear in juxtaposition and may appear in any order.

Actually, this example is somewhat contrived in that the information in the DIMENSION statement could be carried in the INTEGER or the COMMON statement, and the COMMON and DATA statements are illegal together in any but one kind of program. However, the general concept is clear.

Execution Structure

With the exception of the DO statement, executable code in FORTRAN is generally unstructured. The most frequently used conditionals in FORTRAN are the "Logical IF" and the "Arithmetic IF", of which the latter may be regarded as a compressed 3-valued CASE statement. The logical IF is not fully partitioned, in fact, it is specifically "one-tailed". Moreover, the "true" branch may contain only one statement, and that neither a DO nor an IF. Thus, nested IFs are not possible and, even when a nest is not required, tortuous formulations like the following are not uncommon:

```
IF A .EQ. B. AND. B.GT.C GOTO 10
GOTO 30
10 DO 20 I = ...
20 CONTINUE
```

Other conditionals in FORTRAN are the "Assigned GOTO" and the "Computed GOTO"; these are somewhat specialized variants of the CASE statement.

The iteration-control DO statement always excepted, the absence of any kind of grouping construct (such as the PL/I BEGIN...END) is particularly notable.

Procedures

Procedures in FORTRAN, except for the MAIN program, are divided into functions and subroutines, differentiated by mode of invocation. Functions are invoked by a functional notation, e.g., A = B + FUNCT (C) and subroutines by a CALL statement CALL SUBR (A,B,C).

The only internal procedure allowed in FORTRAN is the statement function, defined in a single statement almost identical in form to a normal assigned statement. Any other procedure must be defined externally to the program unit in which the reference to the procedure occurs. A program unit is that sequence of FORTRAN statements which defines a MAIN program, a function subprogram, and a subroutine subprogram. It is important to note that throughout FORTRAN it is tacitly assumed that the program unit is also the unit of compilation, and further that the compiler has in general no knowledge of anything outside the program unit. However, in order to facilitate checking of formal and actual parameters, the following classification has arisen:

a. Statement Functions: Type of parameters and result known to the compiler when compiling the program unit which contains the function definition.

b. Intrinsic Functions: A set of predefined functions - the type of parameters and results are always known to the compiler.

c. Basic External Functions: Almost identical to intrinsic functions - the distinction disappears in FORTRAN76 without apparent incompatibility.

d. External Functions: Usually user-defined functions - the type of result may be known if declared in the invoking program. There is no provision for declaring the type of formal parameters in the invoking program.

e. External Subroutines: The type of formal parameters of external subroutines is unknown in the invoking program.

Scope

The normal scope of reference in FORTRAN is the program unit (see c above). The language definitions do not specify nor allow the specification of scope of allocation, which is usually assumed to be coterminous with scope of reference, although this is often untrue in practice.

The exceptions to the general rule (scope of reference equals program unit) are as follows:

a. Scope of reference of formal parameters of a statement function is the statement function definition.

b. Scope of reference of entities contained in named COMMON blocks is all program units within an executable program (see c below) which contain a definition of that named COMMON block.

c. Scope of reference of entities contained in blank COMMON blocks is the executable program. An executable program is a MAIN program together with all directly or indirectly referenced functions and subroutines.

d. A further scope, not contained in any standard language definition, has been added in some implementations. This scope, usually called GLOBAL, is ALL executable programs containing the definition.

Miscellaneous

Immediately apparent characteristics of FORTRAN are the following:

a. Blanks are syntactically non-significant.
b. There is no statement delimiter except end-of-line.
c. Comments may appear only as separate lines.
d. Keywords are not reserved.
e. Statement identifiers are numeric.
f. Both assignment and DO statements use =.

Unreserved keywords pose a non-trivial problem in parsing FORTRAN programs. Also, the combination of this with other factors results in visual near-ambiguities such as the following:

```
DO 30 I = 1.10  is a valid assignment statement;

while DO30I = 1,10  is a valid DO statement.
```

Section IIIb - FORTRAN Comparative Evaluation

The following are paragraph-by-paragraph comparisons of the TINMAN requirements to the candidate language specifications documents:

A1.FORTRAN.Typed Language
Partially Satisfies

FORTRAN is a typed language but types may be acquired by default in FORTRAN66 (5.3).

FORTRAN76 is similar (4.1.2); in addition the IMPLICIT statement (8.5) may be used to specify the default implied type. "Hollerith" data is allowed in literals; there is no corresponding type for variables.

Necessary language modification: eliminate acquisition of type by default; eliminate Hollerith data type.

A2.FORTRAN.Data Types
Partially Satisfies

FORTRAN66 has types for integer (4.2.1), floating-point (4.2.2), and logical (4.2.5). It has no types for fixed-point or character, though FORTRAN76 has a character type (4.8). FORTRAN66 has also double-precision floating-point (4.2.3), complex (4.2.4), and Hollerith (4.2.6; for data only). FORTRAN66 has arrays of up to 3 dimensions (7.2.1.1). FORTRAN76 allows 7 dimensions (5.4.3). FORTRAN has no concept of records as understood in TINMAN; the EQUIVALENCE statement is sometimes used to apprroximate the effect.

Necessary language modifications: elimination of double-precision, complex, and Hollerith types. Addition of fixed-point (see A4) and character (see A5) types. Removal of restrictions on arrays is linguistically simple. Addition of composite data structures (records) together with unrestricted structures (TINMAN:D5) implies a structured data declaration (cf PL/I:DCL) which in turn suggests the elimination of FORTRAN type statements (e.g., INTEGER) and other declaration statements (e.g., DIMENSION). Implementation considerations: a substantial change is necessary to accommodate unrestricted structures. In addition, most implementations are tied to a fixed maximum number of dimensions in arrays: a change would be non-trivial.

A3.FORTRAN.Precision (floating-point)
Fails to Satisfy

FORTRAN66 allows no precision declaration except REAL and DOUBLE PRECISION which are not defined (4.2.3) (except relative to each other). This is also true of FORTRAN76 (4.5). There is no provision for user declaration of the precision of arithmetic.

Necessary language modifications: allow precision declarations for variables and execution scope. Relatively minor syntactically, though the semantics of combinations of precision declarations for variables and scopes would have to be carefully defined. Implementation considerations are relatively minor.

A4.FORTRAN.Fixed-Point Numbers
Fails to Satisfy

FORTRAN has no provision for fixed-point numbers (except integers) as a computational type, though a fixed-point representation of floating-point numbers is allowed on input/output.

Necessary language modifications: definition of fixed-point type including range and step size. Elimination of separate integer type. These are relatively minor, especially as an adjunct to structure definition (A2). Implementation considerations: code generation for all ranges and step-sizes of fixed point to be designed and implemented. Special casing for integers would be required.

A5.FORTRAN.Character Data
Fails to Satisfy

FORTRAN66 has neither character data nor definition by enumeration. FORTRAN76 has character data (4.8) but no definition by enumeration. Moreover, FORTRAN76 has character strings, concatenation, and a substring notation (4.8; 5.7) which are apparently not required by TINMAN and should be eliminated. The collating sequence of character data in FORTRAN76 is not fully defined (3.1.5), apparently to allow both ASCII and EBCDIC to be standard.

Necessary language modifications: definition by enumeration is sufficient; this is considered below (E6). Apparently the authors of TINMAN are content that ordered collections of characters be representable only as arrays. The character data is sufficiently individual to warrant the special notation of strings, substrings, and concatenation.

A6.FORTRAN.Arrays
Partially Satisfies

FORTRAN66 (7.2.1.1; 7.2.1.1.2) requires specification of the number of dimensions, the type, and the subscript range -- the lower bound always being 1. The upper bound of an array which is a parameter to a procedure may be specified either as a further parameter to the procedure or as a constant value within the procedure; however, space for the actual array is allocated at compile time in the invoking procedure. There is no dynamic space allocation; in general, all space may be considered allocated at either compile or load time.

FORTRAN76 (5.1.1.2) allows specification of subscript range as any contiguous sequence of signed integers. There is no provision in either version of FORTRAN for subscript values which are not integers.

Assuming that FORTRAN76 rules to subscript range are adopted, the extension to allow subscript values to be a subsequence from an enumeration type is relatively minor, given definition of types by enumeration as discussed under E6. Space allocation during execution is a concept foreign to FORTRAN, and necessitates the introduction of storage classes - a somewhat radical change (see also F1).

A7.FORTRAN.Records
Fails to Satisfy

FORTRAN has no concept of records (composite data structures).

Given a proper definition of the language changes to allow specification of composite data structures as discussed under A2, no further modification should be necessary to meet this requirement.

B1.FORTRAN.Assignment and Reference
T (Satisfies)

Assignment and reference are defined for all FORTRAN data types. No FORTRAN data type manages its own storage. FORTRAN has no union types.

B2.FORTRAN.Equivalence
Partially Satisfies

The operator .EQ. in FORTRAN66 is defined only in the relational expression X.EQ.Y where X and Y are numeric expressions of the same data type (6.2). In FORTRAN76 X and Y may be numeric of any type (6.3.2), or they may both be character expressions (6.3.4).

Necessary language modifications: extension of the semantics of the EQUIVALENCE operator to cover all data types. Implementation considerations: relatively minor change required.

B3.FORTRAN.Relationals
T (Satisfies)

Relational operators are defined in FORTRAN66 between numeric expressions of the same type (6.2) and in FORTRAN76 between numeric expressions of any type (6.3.2) and between character expressions (6.3.4). FORTRAN has no types defined by enumeration.

Necessary language modifications: definition of data types by enumeration is discussed under E6.

B4.FORTRAN.Arithmetic Operations
Partially Satisfies

The operators +, -, *, / are defined for all FORTRAN data types (6.1). Exponentiation (**) is defined for certain type combinations. FORTRAN66 has no fixed-point data type.

Necessary language modifications: elimination of COMPLEX data type. Addition of fixed-point data type (see A4). Implementation considerations are minor.

B5.FORTRAN.Truncation and Rounding
Partially Satisfies

FORTRAN has no fixed-point data type and no provision for specification of ranges for integer and floating-point data (beyond the undefined relationship between REAL and DOUBLE PRECISION). Thus, the range is implicit in the implementation; however, no high-order truncation is implicit in any language rule.

No language modifications are necessary, except as discussed under A3 and A4.

B6.FORTRAN.Boolean Operations
Partially Satisfies

FORTRAN66 has no "exclusive-or" operator (6.3); neither does

FORTRAN76 (6.4.1). FORTRAN66 allows but does not dictate short circuit evaluation (6.4); similarly FORTRAN76 (6.6.1). However, most implementations perform short circuit evaluation. Note: "NOR" was used in TINMAN when XOR was intended. (Reference 10 November 1976 meeting at IDA)

Addition of the XOR operator is a trivial change. A language rule dictating short circuit evaluation would raise no problems.

B7.FORTRAN.Scalar Operations
Fails to Satisfy

FORTRAN has no array or structure operations; all operations are on array elements.

The addition to the language of array and structure operations is linguistically simple. Compilation of these operations requires, of course, substantial additions to any current implementation.

B8.FORTRAN.Type Conversion
Partially Satisfies

In FORTRAN66, implicit type conversion takes place across assignment (7.1.1.1) but not otherwise (6.1). Explicit conversion operations among numeric data types are provided (Table 3). Most implementations, however, are more permissive and provide implicit conversion in many mixed-type expressions. FORTRAN76 explicitly defines a number of situations in which implicit conversion must take place.

Elimination of implicit type conversion can be effected by simply adopting a language rule to that effect. Diagnosis of violation of the rule would be required of an implementation.

B9.FORTRAN.Changes in Numeric Representation
Partially Satisfies

No explicit conversion is needed between REAL and DOUBLE PRECISION. The greatest part of this requirement concerns fixed-point data, and is discussed, except for the run-time condition, under B5.

Necessary language modifications: add syntax for run-time exception condition. Depending on the model chosen this could have far-reaching consequences since FORTRAN has no provision for such

conditions. Implementation considerations are problematical, but see also B5.

B10.FORTRAN.I/O Operations
Partially Satisfies

FORTRAN66 has the READ and WRITE statements (7.1.3) which interact with symbolic units which may include devices of all kinds. Data may be sent and received, but not control information. FORTRAN76 has considerably more elaborate I/O facilities (12) with provision for passing some kinds of control information. This requirement is not very specific, but it seems certain that the FORTRAN I/O language is both more and less than is required.

Necessary language modifications: uncertain; dependent on elaboration of requirements.

B11.FORTRAN.Power Set Operations
Fails to Satisfy

FORTRAN has no data types defined by enumeration, no data types defined as power sets, and no operations defined on power sets.

Necessary language modifications: given data types defined by enumeration (E6) provision for definition of power sets and of operations on those sets, presents no substantial difficulties. PL/I bit strings might provide the model. Implementation considerations: substantial but not prohibitive addition.

C1.FORTRAN.Side Effects
T (Satisfies)

FORTRAN66 (6.4) outlaws side effects arising from function evaluation; expressions containing integer division must be evaluated left-to-right. FORTRAN76 (6.5.1) similarly outlaws side effects within a statement. Evaluation proceeds left-to-right within a precedence level, except for exponentiation which is right-to-left (i.e., A**B**C means A**(B**C)).

C2.FORTRAN.Operand Structure
T (Satisfies)

Operator hierarchy in FORTRAN is simple and well-defined. (FORTRAN66: 6.1 through 6.4; FORTRAN76: 6.1, 6.4, 6.5)

C3.FORTRAN.Expressions Permitted
Partially Satisfies

FORTRAN66 allows only a restricted set of expressions as subscripts (5.1.3.3); expressions are not allowed as array declarators (7.2.1.1); expressions are not allowed as DO parameters (5.4.2). FORTRAN76 allows unrestricted expressions as subscripts, integer expressions as array declarators (5.1.2), and unrestricted expressions as DO parameters (11.6).

Adoption of FORTRAN76 rules should satisfy this requirement linguistically. No change would be required to many current implementations which already use FORTRAN76 rules.

C4.FORTRAN.Constant Expressions
Partially Satisfies

FORTRAN66 allows only single constants in subscripts (5.1.3.3), as components of array declarators (7.2.1.1), and as non-variable DO parameters (7.1.2.8). FORTRAN76 allows constant expressions in all these cases (5.4.2; 5.1.2; 11.6). The FORTRAN language definitions do not address compile-time evaluation of expressions; this is therefore left to the implementer's discretion.

Adoption of FORTRAN76 rules should satisfy this requirement linguistically. A rule requiring compile-time evaluation could also be adopted though many would consider this not a proper language requirement. No change would be required to many current implementations which already use FORTRAN76 rules, and generally evaluate obvious constant expressions at compile time. There is room for argument as to what constitutes an "obviously constant" expression.

C5.FORTRAN.Consistent Parameter Rules
T (Satisfies)

FORTRAN parameter rules are consistent.

C6.FORTRAN.Type Agreement in Parameters
Partially Satisfies

Both FORTRAN66 (8.1.2; 8.4.2; 8.3.2) and FORTRAN76 (15.4.2; 15.5.2.2; 15.6.2.3) require formal and actual parameters to agree in type, but FORTRAN76 admits one exception: an actual parameter to a subroutine (as distinct from a function) may be a Hollerith constant (8.4.2). Since a formal parameter cannot be defined to be of Hollerith type, no type agreement is possible in this case. The Hollerith data type has been deleted from FORTRAN76 (21.), so this exception no longer exists. However, since rules are given for the extension of FORTRAN76 to include Hollerith data, thereby reinstating the exception (21.7), the distinction between the two language definitions is not profound.

To provide full compliance with the letter of this requirement only elimination of the Hollerith data type is required - obviously a trivial change. The spirit of the requirement, however, requires not only statement of the rule for type agreement but its enforcement. Traditionally, the "domain of compilation" of a FORTRAN translator has been the program unit; that is, a main program, subroutine, or function. Within a program unit, the only procedure which can be both defined and referenced is the statement function. Type checking therefore can be, and routinely is, performed on parameters to statement function invocations. Formal parameters of other procedures invoked within a program unit are, in general, unknown to the translator and thus cannot be checked. Exceptions to this are intrinsic functions and basic external functions (FORTRAN66 terminology). These are language-defined procedures, the characteristics of whose formal parameters are built-in to the translator for the precise purpose of allowing type checking.

Complete type checking can clearly be implemented by widening the domain of compilation to include the complete executable program. No language changes are made absolutely necessary by this, but a number of inherently desirable changes become feasible:

1. Elimination of the distinction between internally- and externally-defined procedures.

2. Elimination of the not very useful statement functions as a special class.

3. Elimination of the classes "intrinsic functions" and "basic external functions".

Complete type checking would be a substantial change to any current implementation.

C7.FORTRAN.Formal Parameter Kinds
Fails to Satisfy

FORTRAN has no concept of parameter classes, though some difficulties with procedure parameters are recognized.

Parameter classes and usage rules must be established. This presents no great syntactic difficulty. Checking during compilation for usage consistent with classification is not trivial.

C8.FORTRAN.Formal Parameter Specifications
Fails to Satisfy

FORTRAN contains no generic procedure capability.

Allowing optional specification of parameter attributes is a minor language change. "Instantiating" a generic procedure at compile time requires non-trivial implementation work.

C9.FORTRAN.Variable Numbers of Parameters
Fails to Satisfy

Procedures with a variable number of arguments cannot be defined in FORTRAN; though certain intrinsic functions with this characteristic are pre-defined and invokable from a FORTRAN program (e.g., MAX, MIN).

Necessary language modifications: allow a variable number of arguments in procedure definitions. There are no serious implementation consideratons.

D1.FORTRAN.Constant Value Identifiers
Fails to Satisfy

Constants cannot be associated with identifiers in FORTRAN66. FORTRAN76 provides the PARAMETER statement for this purpose.

Necessary language modifications: adopt the PARAMETER statement or some other mechanism. The implementation considerations are minimal.

D2.FORTRAN.Numeric Literals
T (Satisfies)

FORTRAN provides a syntax for constants of all built-in data types.

Interpretation of numeric constants is an implementation rather than a language consideration.

No language changes are necessary, and current implementations (as distinct from older implementations) usually provide consistent conversions. Note that assembler conversions are also relevant.

D3.FORTRAN.Initial Values of Variables
T (Satisfies)

Variables, except those in blank COMMON, may be initialized by use of the DATA statement. There are no language-defined default values. The allocation scope is either the named COMMON block or the program unit. Testing for initialization is not a language consideration.

No language changes are necessary, and incomplete compile-time testing for initialization is feasible at reasonable cost; however, run-time testing is potentially very costly.

D4.FORTRAN.Numeric Range and Step Size
Fails to Satisfy

There is no provision in FORTRAN for numeric range specification.

Necessary language modifications: allow range and step-size declarations. These are relatively minor changes given the changes necessary under A2, A4, etc. Implementation costs are variable depending on the degree of run-time checking provided.

D5.FORTRAN.Variable Types
Fails to Satisfy

FORTRAN allows only scalars as array elements, and allows no structures other than arrays.

No syntax changes are required in data specification beyond those for definition of composite data structures (A2), if these are properly defined. There may be a problem devising a tidy notation for referring to, e.g., arrays as elements of an array (perhaps X(I,J(K,L))?).

There is no question that generalizing structures as required here would have a major impact on any current FORTRAN implementation. The

internal tables of FORTRAN compilers tend to be constructed to deal efficiently with the restricted attributes of FORTRAN arrays (maximum number of dimensions, scalar elements). The necessary redesign of these tables, in addition to the code generation changes, amounts to a redesign of almost any conceivable FORTRAN compiler.

D6.FORTRAN.Pointer Variables
Fails to Satisfy

FORTRAN has no provision for pointer variables.

Necessary language modifications: relatively simple syntactically (models exist), but with substantial implementation considerations.

E1.FORTRAN.User Definitions Possible
Partially Satisfies

FORTRAN provides no capability for definition of data types and operations, except by definition of functions and subroutines which can be construed as new operations.

A necessary preliminary modification is the provision of a simple uniform grammar (H1). This, and the addition of user-definition facilities, constitute a radical change, amounting to the elimination of most distinguishably FORTRAN characteristics. Implementation considerations are radical. It is doubtful if any current implementation could absorb it.

E2.FORTRAN.Consistent Use of Types
Fails to Satisfy

FORTRAN allows no defined types.

Necessary language modifications are described under E1.

E3.FORTRAN.No Default Declarations
Fails to Satisfy

FORTRAN allows explicit declarations but provides certain defaults.

Necessary language modifications: eliminate default declarations.

E4.FORTRAN.Can Extend Existing Operators
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

Necessary language modifications are discussed under E1.

E5.FORTRAN.Type Definitions
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

The modifications discussed under E1 can easily be changed to meet this requirement.

E6.FORTRAN.Data Defining Mechanism
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

The modifications discussed under E1 can include the required mechanisms without difficulty.

E7.FORTRAN.No Free Union or Subset Types
T (Satisfies)

FORTRAN satisfies this by default since there is no capability for definition of data types.

Modifications discussed under E1 can also be defined in order not to conflict with this requirement.

E8.FORTRAN.Type Initialization
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

Modifications discussed under E1 can be defined to meet this requirement.

F1.FORTRAN.Separate Allocation and Access Allowed
Fails to Satisfy

There is no user facility in FORTRAN for distinguishing between scope of allocation and scope of reference. In FORTRAN66, the distinction is not recognized. In FORTRAN76, the SAVE statement (8.9) allows a distinction to be made in an indirect manner.

The introduction of storage classes or a similar concept is a necessary language modification. This ties in with the modifications discussed under C6 and C7. Depending on the model chosen, the necessary modifications might well be radical.

F2.FORTRAN.Limiting Access Scope
Fails to Satisfy

FORTRAN, with the exception of procedures, has no concept of accessing separately-defined entities.

Necessary language modifications: introduction of explicit name-scoping rules. This would be a radical extension in conjunction with related changes under C6, C7, and F1.

F3.FORTRAN.Compile Time Scope Determination
T (Satisfy)

The scope of all FORTRAN identifiers is wholly determined at compile time. Thus, taken in isolation, this requirement is satisfied by FORTRAN. However, see F1 and F2.

There are no necessary language modifications, but see F1 and F2.

F4.FORTRAN.Libraries Available
Partially Satisfies

A library of procedures is available to FORTRAN users. There is generally no access at compile time. Data definitions are not accessible.

Modifications necessary to meet this requirement form part of those discussed under F5.

F5.FORTRAN.Library Contents
Fails to Satisfy

FORTRAN has no provision for a compile-time accessible library (unless the availability of the interface specifications of intrinsic functions is considered a partial exception).

A facility like the PL/I %INCLUDE would meet this requirement. Implementation of such a facility is not difficult, though non-trivial. Access to, and checking of, interface specifications is part of the work considered under C6 and C7.

F6.FORTRAN.Libraries and Compools Indistinguishable
Fails to Satisfy

FORTRAN provides no compile-time accessible library. However, given the change discussed under F5, there seems no reason to distinguish libraries and compools.

No language changes are necessary beyond those mentioned under F5.

F7.FORTRAN.Standard Library Definitions
Partially Satisfies

FORTRAN66 provides standard machine-independent interfaces to input/output devices. These would probably not be regarded as adequate by the authors of TINMAN, since the nature of the interfaces is largely dictated by first-generation ranking. FORTRAN76 provides more up-to-date interfaces, though still inadequate. Some implementations provide subroutine interfaces to hardware clocks, etc., though these cannot be considered standard.

Specification of language would necessitate careful consideration of the full range of devices, special hardware, and other external facilities to be supported. Implementation to the defined interfaces should not be difficult.

G1.FORTRAN.Kinds of Control Structures
Partially Satisfies

FORTRAN provides mechanisms for sequential, conditional and iterative control. It provides none for recursion, parallel processing, exception handling and asynchronous interrupt handling.

Necessary language modifications: see later requirements in this section (G3 through G8).

G2.FORTRAN.The GOTO
T (Satisfies)

The destination of the FORTRAN GOTO is limited to the program unit, which is the most narrow scope defined in FORTRAN (with the exception of the statement function, which cannot contain a GOTO).

G3.FORTRAN.Conditional Control
Fails to Satisfy

FORTRAN provides no fully-partitioned conditional control. The "arithmetic IF" (FORTRAN66: 7.1.2.2) is more properly a 3-valued CASE statement; the "logical IF" (FORTRAN66: 7.1.2.3) is specifically "one-tailed" (no ELSE branch). Moreover, only one statement (and that neither a DO nor a logical IF) may be executed on the "true" branch. Thus, nesting of conditionals is impossible. The "Assigned GOTO" is another variant on a CASE statement (FORTRAN66: 7.1.2.1.2): a numeric statement identifier is "assigned" to an integer variable whose value is tested at the decision point. The "Computed GOTO" is a more orthodox CASE statement (FORTRAN66: 7.1.2.1.3): the value of an integer variable is used as an index to a list of statement identifiers.

Necessary language modifications: provide IF...THEN...ELSE as described in TINMAN; eliminate Logical and Arithmetic IF; eliminate Assigned GOTO; and provide grouping control (e.g., BEGIN...END).

G4.FORTRAN.Iterative Control
Partially Satisfies

The FORTRAN DO permits entry only at the head; but the DO-variable is not local to the loop, and termination is not allowed in the middle of the loop. FORTRAN66 evaluates the termination condition at the end of the loop, while FORTRAN76 evaluates it at the head of the loop (see FORTRAN76: 20.11).

Necessary language modifications: redesign loop control as required.

G5.FORTRAN.Routines
Fails to Satisfy

FORTRAN has no provision for the definition of recursive routines.

Necessary language modifications: though the provision of the ability to define recursive routines is syntactically not difficult, the implications are radical in the context of FORTRAN, and amount to a redefinition of the language. See C6, C7, F1, F2.

G6.FORTRAN.Parallel Processing
Fails to Satisfy

FORTRAN has no provision for parallel processing. The subroutine package devised by the Instrumentation Society of America (ISA) and augmented variously by implementers provides a partial solution.

Necessary language modifications: provide task identifiers, scheduling mechanisms, etc. See also G8. The underlying mechanisms required for implementation is a substantial piece of work for each object machine.

G7.FORTRAN.Exception Processing
Fails to Satisfy

FORTRAN provides no mechanism for exception handling, except that certain well-defined input/output exceptions are provided for in FORTRAN76 (12).

Necessary language modifications: provide a modified PL/I ON-unit capability or its equivalant. These are not trivial changes.

G8.FORTRAN.Synchronization and Real-Time
Fails to Satisfy

FORTRAN has no provision for real-time; the ISA subroutine package provides some of the required facilities (see G6).

Language modifications required: in addition to modifications in G6, provide the ability to delay tasks, specify priorities, accept asynchronous interrupts, and access real-time clocks. Relatively sophisticated control must be provided for each object machine.

H1.FORTRAN.General Characteristics
Partially Satisfies

FORTRAN66 is almost free format (except that columns 1-6 of the input line are reserved for statement identifiers, comments and continuation codes). It has no explicit statement delimiter, allows mnemonically significant (but short) names for data, but only numeric statement identifiers; has a simple, easily parsed, but not uniform grammar; has no unique notations (depending on the definition), allows no abbreviations, and admits to no syntactic ambiguities.

Language modifications required: eliminate special significance of columns 1-6; add explicit statement delimiter; eliminate grammatical non-uniformities (a substantial list); allow non-numeric identifiers of arbitrary length for both statements and variables; and make keywords reserved to simplify parsing. Note that these changes, though apparently simple, would in themselves make the resulting language barely recognizable as FORTRAN. Implementation considerations are difficult to assess in isolation. The effects of these changes and those discussed elsewhere are enough to transform the language, and thus the implementation, entirely.

H2.FORTRAN.No Syntax Extensions
T (Satisfies)

No syntax extensions are permitted in FORTRAN.

H3.FORTRAN.Source Character Set
T (Satisfies)

Any FORTRAN66 program may be written using only 47 characters (which are all contained in the ASCII subset) (3.1). FORTRAN76 (3.1.4) adds the apostrophe and the colon for a total of 49.

H4.FORTRAN.Identifiers and Literals
Partially Satisfies

FORTRAN66 provides formation rules for identifiers and numeric literals (3.4; 3.5; 5.1.1) and FORTRAN76 rules are identical. No break character is specified. Blanks or spaces are ignored and may be used for clarity. FORTRAN66 gives rules for Hollerith constants (5.1.1.6). FORTRAN76 gives rules for character literals: there is no break character and spaces are significant. Separate quoting is neither required nor allowed; long literals (Hollerith or character) must be continued at the beginning of the next line.

Language modifications required: provide break character for identifiers and literals and possibly eliminate non-significance of

blanks. Provide a separate-quoting convention and require its use for long literals by disallowing continuation of literals over end-of-line.

H5.FORTRAN.Lexical Units and Lines
Fails to Satisfy

Lexical units in FORTRAN may be continued across lines at will, with the exception of the END statement (occurs once in a program unit) which must be contained on a single line. Object characters are not defined in the FORTRAN language; the practical effect is that they may in most cases safely be included in literal strings, but there is no guarantee that such usage may not conflict with the internal conventions of a particular translator.

Adoption of a rule against continuing lexical units across lines is easy, but enforcement of the rule would be a burden for a translator without further changes. In particular, the non-significance of blanks, unreserved keywords, and the default definition of data names, make impossible the immediate recognition of a lexical unit by a translator. Elimination of these factors (which except for the first is explicitly required elsewhere in TINMAN), would ease the burden considerably. In conjunction with this rule, definition of a literal string as an instance of a lexical unit which must not be continued across end-of-line (as required by H4) would seem to be sufficient to allow safe inclusion of object characters in the string.

H6.FORTRAN.Key Words
Fails to Satisfy

FORTRAN key words are not reserved. By rough count there are 32 keywords in FORTRAN66; this does not include logical (3) and relational (6) operators, FORMAT field descriptors (9), and names of intrinsic and basic external functions (54). It also excludes arithmetic operators and other basic syntactical symbols (parentheses, comma, slash, =).

Reserving key words is simple and in accordance with good FORTRAN programming practice. The benefit would be felt by the implementers, who could eliminate the absurd double-parsing necessary to recognize that the statement "DO 30 I = 1.10" is a perfectly legitimate assignment statement by existing FORTRAN rules. The statement "DO30I = 1,10" is equally a legitimate DO statement. Here again as in H5, the non-significance of blanks must be abolished to realize the full intent of reserving key words.

H7.FORTRAN.Comment Conventions
Fails to Satisfy

Comments may appear only on separate lines. They are introduced by the character C in column 1 of such a line in FORTRAN66 (3.2.1). In FORTRAN76 the character * may be substituted (3.2.1).

Adoption of a more flexible comment convention, such as the PL/I /*...*/ pair, should introduce no difficulties.

H8.FORTRAN.Unmatched Parentheses
T (Satisfies)

FORTRAN formally allows no unmatched parentheses and so satisfies this requirement. However, use of parentheses in FORTRAN is strictly local - a left parenthesis must be matched by a right parenthesis within the same statement. Parenthetical constructs in the larger and more important sense - such as the BEGIN...END pair - do not exist in FORTRAN, although some such construct would have to be introduced if FORTRAN were to be modified to meet TINMAN requirements.

H9.FORTRAN.Uniform Referent Notation
Partially Satisfies

There are restrictions on subscript forms in FORTRAN66. Elsewhere, TINMAN requires the elimination of such restrictions and they are eliminated in FORTRAN76. Given that elimination, there is no syntactical distinction in FORTRAN between a function reference and an array reference, so the requirement is satisfied with respect to arrays. (No data structures other than arrays are defined in FORTRAN.) As to scalar data, technically the question does not arise in FORTRAN66 which does not recognize the possibility of parameterless functions, but requires an invocation of such a function to include an empty parenthesis as a parameter list (15.5.1, 15.2.1). Hence, a reference to scalar data may not be replaced without change by a reference to a parameterless function.

Given that restrictions on subscript forms are eliminated, elimination of empty parameter lists is all that is required to make FORTRAN satisfy this requirement with respect to existing data structures). Since other composite data structures are also a rquirement (A2), the referent notations for such structures must be defined appropriately.

H10.FORTRAN.Consistency of Meaning
Fails to Satisfy

The operator '=' functions both as an assignment operator and in

the specification of DO parameters. A statement function definition (FORTRAN66: 8.1.1) is syntactically indistinguishable from a normal assignment statement.

Language modifications required: respecify iteration control and eliminate statement function.

I1.FORTRAN.No Defaults in Program Logic
Fails to Satisfy

The following is a selection of instances of defaults from FORTRAN66 (there are many others): (1) action on encountering end-of-file on READ is undefined (7.1.3.3.3), (2) action at Computed GOTO defined only for values within range (7.1.2.1.3), (3) loop control variable is undefined on loop exit (7.1.2.8.1), and (4) precision of floating-point data is implementer-defined (4.2.3). FORTRAN76 corrects the first three cases (12.7; 11.2; 11.6.7) and many others. However, it does not define precision of floating point, and only partially specifies the collating sequence of character data.

Adoption of FORTRAN76 rules would correct many of the defects in FORTRAN66. Correction of the explicit FORTRAN76 omissions would further improve the language. However, without painstaking study there is no way of knowing how many implicit defaults still remain.

I2.FORTRAN.Object Representation Specification Optional
Fails to Satisfy

FORTRAN66 does not specify object representations which are therefore implementer-defined. There is no override capability. FORTRAN76 defines the relative sizes of storage units for defined data types (17.1.1) and defines the ordering of arrays (5.4.2) but is otherwise silent on object representation. Again, there is no override capability.

Language modifications required: specify default object representation and provide override capability.

I3.FORTRAN.Compile Time Variables
Fails to Satisfy

There is no provision for environmental inquiry in FORTRAN, though a number of implementations have allowed user specification. However, the action taken was usually implementer-defined.

Language modifications required: provide for interrogation of compile-time variables.

I4.FORTRAN.Conditional Compilation
Fails to Satisfy

FORTRAN does not define any conditional compilation capability.

Language modifications required: provide conditional compilation capability.

I5.FORTRAN.Simple Base Language
Partially Satisfies

FORTRAN66 is a simple language, however, with some redundancies; e.g., (1) Computed GOTO and Assigned GOTO, (2) logical and arithmetic IF, and (3) ability to specify arrays in either DIMENSION or type statements.

FORTRAN, even with redundancies removed, seems hardly suitable as a base for extension.

I6.FORTRAN.Translator Restrictions
Partially Satisfies

FORTRAN66 specifies the size of identifiers and the maximum number of array dimensions, but does not specify number of identifiers or other factors relevant to translator design, which therefore becomes implementer-defined.

Language modifications required: specify suitable translator requirements.

I7.FORTRAN.Object Machine Restrictions
Partially Satisfies

FORTRAN defines no object machine restrictions. An implementation, however, is not prevented from doing so.

Language modifications required: declare arbitrary restrictions illegal.

J1.FORTRAN.Efficient Object Code
T (Satisfies)

FORTRAN, of course, can be very efficiently implemented. However, the language as modified to meet a substantial subset of the TINMAN requirements would present a more difficult problem.

J2.FORTRAN.Safe and Effective Optimization Possible
Partially Satisfies

FORTRAN is simple enough to allow safe optimization in most cases. Some problems remain. For instance, although it is forbidden for a statement to contain a function reference which has side effects upon the containing statement, there is no way of ensuring that this is the case.

Language modifications required: define precise conditions which must be met before optimization is allowed.

J3.FORTRAN.Machine Language Insertions
Fails to Satisfy

FORTRAN defines no mechanism for machine-language insertions.

Language modifications required: define a machine-language insertion mechanism.

J4.FORTRAN.Object Representation Specification
Fails to Satisfy

FORTRAN does not support composite data structures or the specification of their object representation.

Language modifications required: support for composite data structures as under A2 and allowance for specification of object representation.

J5.FORTRAN.Open and Closed Routine Calls
Fails to Satisfy

FORTRAN does not define a method of specifying whether calls are

open or closed. While no rule is laid down, the assumption is that called routines are externally defined and are not available for in line inclusion.

Language modifications required: allow for user specification of open or closed calls.

K1.FORTRAN.Operating System Not Required
T (Satisfies)

FORTRAN requires no operating system. Certain features such as input/output, do require run-time support routines. It should be noted that perhaps the full range of TINMAN facilities -- parallel processing, task scheduling, etc. -- requires an array of run-time services which may be hard to distinguish formally from an operating system.

K2.FORTRAN.Program Assembly
Partially Satisfies

Separate definition and separate compilation is the assumed (though not required) method of operation in FORTRAN. It is assumed that the integration of separately compiled modules will be performed by a separate mechanism (e.g., link editor) rather than under the control of the translator.

Language modifications required: the integration of modules by the translator is a minor variant on the "domain of compilation" changes for type-checking, library support, etc. discussed under C6, C7, F1, F2.

K3.FORTRAN.Software Development Tools
T (Satisfies)

This is not entirely a language requirement; the FORTRAN language definition does not address software development tools. However, as a widely used scientific programming language, FORTRAN has many tools available.

K4.FORTRAN.Translator Options
Unknown

This is not a language requirement. The FORTRAN language definition does not address translator options.

K5.FORTRAN.Assertions and Other Optional Specifications
Fails to Satisfy

FORTRAN makes no provision for optionally interpretable comments.

Definition of a method of denoting optionally interpretable comments would be trivial.

L1.FORTRAN.No Superset Implementations
Fails to Satisfy

FORTRAN66 specifically allows superset implementations (Appendix B1) as does FORTRAN76 (1.3).

Modify specifications to forbid superset implementations.

L2.FORTRAN.No Subset Implementations
T (Satisfies)

Subset implementations of FORTRAN66 do not conform to the standard (Appendix B1). A subset language is defined in FORTRAN76. Translators conform to the subset standard if, at the very least, the subset language is provided (1.3.1).

L3.FORTRAN.Low-Cost Translation
T (Satisfies)

Many implementations bear witness that low-cost translation of FORTRAN is possible. However, translation of FORTRAN as modified to meet a substantial number of the TINMAN requirements presents a different problem. Note: As an implementation requirement, this is in partial conflict with L4.

L4.FORTRAN.Many Object Machines
T (Satisfies)

Not a language requirement; however, FORTRAN probably has more implementations than any other language. Note: For one translator to produce code for a variety of object machines, a machine-independent intermediate language is required. Design of such a language is a problem almost as formidable as design of a programming language. Further, this precludes one-pass compilation, which is one way of

meeting requirement L3. More subtly, the necessary generality of an intermediate language may obscure correspondences between the original input and the object machine with the effect that more work is done for potentially less efficient results.

L5.FORTRAN.Self-Hosting Not Required
T (Satisfies)

FORTRAN translators have been written for a wide variety of host machines, some of them quite small. Note: It is unlikely that the same design would be used for both a translator for a small-host machine, and a translator for many object machines (L4).

L6.FORTRAN.Translator Checking Required
Fails to Satisfy

This is not a language requirement (but see C6). (No language modifications required.)

As discussed under C6, checking of formal against actual parameters necessitates a departure from the traditional FORTRAN concept of the program unit as the "domain of compilation".

L7.FORTRAN.Diagnostic Messages
Fails to Satisfy

The FORTRAN language definition does not prescribe diagnostic messages or translator actions for erroneous programs. FORTRAN66 mentions the possibility of standardizing "diagnostics" but dismisses it as "premature". FORTRAN76 does not discuss it.

Prescribe diagnostic messages and translator action. Note that, as mentioned in FORTRAN 66, prescription of diagnostic messages and translator action may have profound implications for translator design. (Reference Appendix B1)

L8.FORTRAN.Translator Internal Structure
T (Satisfies)

Neither FORTRAN66 nor FORTRAN76 dictates translator characteristics.

L9.FORTRAN.Self-Implementable Language
Fails to Satisfy

Translators have been written in FORTRAN, but only with some special extensions made to the language. As defined, however, FORTRAN is not a suitable language for writing language translators.

If modifications were made to FORTRAN to meet any reasonable number of TINMAN requirements, particularly A2 (Composite Data Structures), the resulting language would be suitable for self-implementation.

M1.FORTRAN.Existing Language Features Only
Partially Satisfies

FORTRAN76 is well within the state of the art, as is, even more so, FORTRAN66. The question remains whether modifications to meet TINMAN requirements could be made within the state of the art. There seems no reason why this could not be accomplished.

M2.FORTRAN.Unambiguous Definition
Fails to Satisfy

There is no formal definition of FORTRAN. A good part of FORTRAN76 is evidence of the semantic ambiguities of FORTRAN66. In this respect, FORTRAN76 might be considered the unambiguous definition of FORTRAN66, though it is not clear how many ambiguities remain. However, it should be pointed out that a formal definition gives only the comforting appearance of infallibility. In reality, it needs to be syntactically and semantically debugged as carefully as any other program.

.. No language modifications; develop formal definition.

M3.FORTRAN.Language Documentation Required
Fails to Satisfy

Innumerable descriptions exist of the FORTRAN language and of particular implementations. It is doubtful whether any meets the requirement as stated.

Provide language documentation as specified.

M4.FORTRAN.Control Agent Required

Fails to Satisfy

This is not a language requirement. DoD is free to control (within its own domain) any language it adopts. As it stands, FORTRAN is controlled by ANSI, and thus there would be difficulties arising from divergent interpretations.

Set up strongly supported compiler validation agency.

M5.FORTRAN.Support Agent Required
Fails to Satisfy

This is not a language requirement.

No language modifications are required. Presumably the validation agency set up under M4 could act as the support agency as well as manage configuration for compilers, tools, libraries and standards.

M6.FORTRAN.Library Standards and Support Required
Fails to Satisfy

This is not a language requirement. FORTRAN does not support libraries in the sense intended by TINMAN (see E1, F4, F5).

Presumably the validation agency set up under M4 could act as the support agency as well as manage configuration for compilers, tools, libraries and standards.

Section IIIc - FORTRAN Language Features Not Needed

#.FORTRAN.Deletions From FORTRAN66 To Meet TINMAN Requirements

As noted elsewhere, modification of FORTRAN to meet TINMAN requirements must result in a major transformation of the language. It is thus difficult to distinguish in all cases between language which should be deleted because its function is not required, and language which should be deleted because it is inadequate to represent the required function. An attempt can however be made:

1. The (admittedly trivial) functions represented by the PAUSE and STOP statements are not required. These statements can therefore be deleted.

2. Under TINMAN I5 ("Simple Base Language"), language features must provide a simple unduplicated capability. Under this rule the COMPLEX data type, together with its associated operations, "intrinsic" functions, and "basic external" functions (CONJG, CSQRT, etc.) must be eliminated: COMPLEX can clearly be represented as a "defined type". Hollerith data is a duplication of the required character data (A2) and should be eliminated. The function of the FORMAT statement is largely an implicit duplication of the required explicit object-to-character conversion (B8): the FORMAT statement should be eliminated. Either the arithmetic or logical IF should be eliminated as a duplicate function; however, neither is adequate to meet the requirements (see below). A precisely similar argument applies to the Computed and Assigned GOTO.

3. The bulk of the eliminations arise from the inadequacy of the FORTRAN language features, as defined. Typically, TINMAN calls for a general capability while FORTRAN traditionally has provided a more restricted capability. Assuming that the general capability is provided, the FORTRAN feature becomes redundant and should be eliminated. These questions are discussed under the particular requirements and may not be strictly relevant to this section; however, it seems appropriate to provide a consolidated list:

Requirement	FORTRAN Eliminated	Comments
A2, A7, D5	Type statements DIMENSION EQUIVALENCE DATA	Unrestricted structures require structured data declaration.
A3	Double-precision data type	Precision specifications rule out arbitrary (machine-dependent) precision.
C6	Statement functions. Intrinsic function. Basic external functions.	Universal type checking suggests elimination distinctions between internal and external function definitions.
F1	COMMON	General mechanism to specify allocation and reference scope of variables rules out the restricted FORTRAN solution
G3	Arithmetic IF Logical IF Computed GOTO Assigned GOTO	These features are inadequate to meet the requirement for fully-partitioned conditionals.
G4	DO	The FORTRAN DO does not meet the stated requirement for an iterative control structure.
H1	Numeric statement identifiers	Identifiers are required to be mnemonic.
H7	Comments convention	Comments must be able to appear "anywhere reasonable". A separate line is not sufficient.

Section IIId - FORTRAN Summary and Recommendations

The following is a table showing the TINMAN requirements by category and the requirements which FORTRAN fully or partially satisfy or failed to satisfy as well as those which are undetermined (unknown).

REQUIREMENT		FORTRAN
A.	Data and Types	
	A1. Typed Language	P
	A2. Data Types	P
	A3. Precision	F
	A4. Fixed-Point Numbers	F
	A5. Character Data	F
	A6. Arrays	P
	A7. Records	F
B.	Operations	
	B1. Assignment and Reference	T
	B2. Equivalence	P
	B3. Relationals	T
	B4. Arithmetic Operations	P
	B5. Truncation and Rounding	P
	B6. Boolean Operations	P
	B7. Scalar Operations - On arrays	F
	B8. Type Conversion - Implicit	P
	B9. Type Conversion - Explicit	P
	B10. I/O Operations	P
	B11. Power Set Operations	F
C.	Expressions and Parameters	
	C1. Side Effects	T
	C2. Operand Structure	T
	C3. Expressions Permitted	P
	C4. Constant Expressions	P
	C5. Consistent Parameter Rules	T
	C6. Type Agreement in Parameters	P
	C7. Formal Parameter Kinds	F
	C8. Formal Parameter Specification	F
	C9. Variable Numbers of Parameters	F
D.	Variables, Literals and Constant	
	D1. Constant Value Identifiers	F
	D2. Numeric Literals	T
	D3. Initial Values of Variables	T
	D4. Numeric Range and Step Size	F
	D5. Variable Types	F
	D6. Pointer Variables	F
E.	Definition Facilities	

E1.	User Definitions Possible	P
E2.	Consistent Use of Types	F
E3.	No Default Declarations	F
E4.	Can Extend Existing Operators	F
E5.	Type Definitions	F
E6.	Data Defining Mechanisms	F
E7.	No Free Union or Subset Types	T
E8.	Type Initialization	F

F. Scope and Libraries

F1.	Separate Allocation and Access Allowed	F
F2.	Limiting Access Scope	F
F3.	Compile Time Scope Determination	T
F4.	Libraries Available	P
F5.	Library Contents	F
F6.	Libraries and Compools Indistinguishable	F
F7.	Standard Library Definitions	P

G. Control Structures

G1.	Kinds of Control Structures	P
G2.	The GOTO	T
G3.	Conditional Control	F
G4.	Iterative Control	P
G5.	Routines	F
G6.	Parallel Processing	F
G7.	Exception Handling	F
G8.	Synchronization and Real-Time	F

H. Syntax and Comment Conventions

H1.	General Characteristics	P
H2.	No Syntax Extensions	T
H3.	Source Character Set	T
H4.	Identifiers and Literals	P
H5.	Lexical Units and Lines	F
H6.	Key Words	F
H7.	Comment Conventions	F
H8.	Unmatched Parentheses	T
H9.	Uniform Referent Notation	P
H10.	Consistency of Meaning	F

I. Default, Conditional Compilation and Language Restrictions

I1.	No Defaults in Program Logic	F
I2.	Object Representation Specifications Optional	F
I3.	Compile Time Variables	F
I4.	Conditional Compilation	F
I5.	Simple Base Language	P
I6.	Translator Restrictions	P
I7.	Object Machine Restrictions	P

J. Efficient Object Representations and Machine Dependencies

J1.	Efficient Object Code	T
J2.	Optimizations Do Not Change Program Effect	P
J3.	Machine Language Insertions	F
J4.	Object Representation Specification	F
J5.	Open and Closed Routine Calls	F
K.	**Program Environment**	
K1.	Operating System Not Required	T
K2.	Program Assembly	P
K3.	Software Development Tools	T
K4.	Translator Options	U
K5.	Assertions and Other Optional Specifications	F
L.	**Translators**	
L1.	No Superset Implementations	F
L2.	No Subset Implementations	T
L3.	Low-Cost Translation	T
L4.	Many Object Machines	T
L5.	Self-Hosting Not Required	T
L6.	Translator Checking Required	F
L7.	Diagnostic Messages	F
L8.	Translator Internal Structure	T
L9.	Self-Implementable Language	F
M.	**Language Definition, Standards and Control**	
M1.	Existing Language Features Only	P
M2.	Unambiguous Definition	F
M3.	Language Documentation Required	F
M4.	Control Agent Required	F
M5.	Support Agent Required	F
M6.	Library Standards and Support Required	F

Totals	T (Fully Satisfies)	21
	P (Partially Satisfies)	27
	F (Fails to Satisfy)	49
	U (Unknown)	1

Recommendation

FORTRAN is not recommended as a base for the DoD HOL. To the reader of TINMAN, it is clear that FORTRAN cannot be modified to satisfy any significant part of the requirements, while maintaining its individuality as a language:

1. FORTRAN's traditional unstructured data specification facilities cannot survive the introduction of composite data structures (A2) especially when fully generalized (D5). A

structured data declaration such as PL/I's DCL is required; DIMENSION, INTEGER, etc. become redundant.

2. Introduction of a fully partitioned conditional control (G3) effectively makes redundant all of FORTRAN's conditional and switching mechanisms (Logical and arithmetic IF; Assigned and Computed GOTO). The IF...THEN...ELSE of PL/I requires only minor change.

3. Complete type checking (C6) widens the FORTRAN "domain of compilation" from the traditional "program unit" to the complete executable program. This suggests relaxing the distinction between internally and externally defined procedures and leads, logically, to the elimination of the familiar FORTRAN function classes: statement functions, intrinsic functions, basic external functions, external functions (which are distinguished from each other mainly by the degree of type checking to be performed by the translator). PL/I satisfies the requirement with only minor change.

4. User specification of allocation and access scope (F1) implies the introduction of storage classes and the elimination of COMMON. PL/I already has storage classes.

5. Reserved keywords (H6), a specific statement delimiter (H1), mnemonic statement identifiers (H1) and the required comments conventions (H7) all require changes to FORTRAN, but none to PL/I.

Thus, by consideration of only a few requirements, we have eliminated nearly all the distinctive features of FORTRAN - all that distinguishes FORTRAN from a rather weak subset of PL/I. By contrast, PL/I requires very little change to meet those same requirements. In fact, there is probably no positive (as distinct from restrictive) requirement that PL/I does not satisfy at least as well as FORTRAN. The conclusion is inescapable: PL/I is a far stronger candidate than FORTRAN, which should be eliminated from contention.

It is understood, of course, that a similar case could be made for ALGOL68, and others, as against FORTRAN.

PL/I has been discussed simply because it used FORTRAN as a base upon which to add functions to satisfy a large class of problems.

Section IV - COBOL EVALUATION

Section IVa - COBOL Language Introduction

#.COBOL.Introduction

The primary evaluation of COBOL with respect to individual TINMAN requirements has been made using ANSI Standard COBOL 74; thus, wherever the word "COBOL" is used it should be understood that it means (only) the ANSI Standard COBOL 74. COBOL 74 is defined by the "American National Standard Language COBOL", ANSI X3.23-1974, American National Standards Institute, New York, NY, 1974.

The evaluation of individual TINMAN requirements with regard to COBOL 68 was done after the COBOL 74 evaluations. There are very few changes in the individual evaluations of Fully, Partially, or Fail due to differences between COBOL 68 and 74. For that reason, no comments about COBOL 68 have been made unless the actual evaluation for a specific requirement has been changed due to differences between COBOL 68 and 74. However, it is important to note that some of the text discussion, and specific illustrations from the language, and the rationale behind the COBOL 74 evaluation does not (necessarily) apply to COBOL 68; only the final evaluation is the same if no mention of COBOL 68 is made. COBOL 68 is defined by the "United States of America National Standard COBOL", ANSI X.23-1968, American National Standards Institute, New York, NY, 1968.

When appropriate, separate paragraphs are provided which indicate the level of difficulty in changing COBOL 74 to satisfy the TINMAN requirements. In considering this aspect, some attention was paid to the "CODASYL 1976 Journal of Development" which represents the current definition of COBOL, although it is not a standard.

FUNDAMENTAL CONCEPTS OF COBOL

Four Divisions:

> The most fundamental concept of COBOL is the existence of four divisions in the language: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. A source program contains information in all four divisions. The IDENTIFICATION Division simply provides the identification of the program and programmer. The ENVIRONMENT Division allows the user to indicate a great many features which relate to the hardware, and permits the connection between the source program and the hardware.

Almost by definition the ENVIRONMENT Division for a particular program would have to be rewritten if the program was moved to another machine. The intent of the ENVIRONMENT Division is to allow all hardware-dependent elements to be encapsulated in one place.

The DATA Division contains facilities to describe logical files and logical records and allows some user control over the object-time representation of the data. This is a greatly expanded and powerful version of the concept of "data declarations" in other languages. This division is relatively machine-independent, but not entirely so if efficiency of the internal data storage is a major consideration (which is normally the case). Thus it may be necessary to rewrite this division in going from one machine to another, although that can be avoided at the expense of some object-time efficiency. A single DATA Division can be written and used for any source program which uses the same files.

The PROCEDURE Division allows the user to specify the operations which are to be carried out in a particular program. This corresponds to most of what is normally thought of as "the program" in other languages. The PROCEDURE Division is machine-independent providing the user does not deliberately go out of his way to do things which are machine-dependent, and avoids the few elements in this division which are hardware-dependent (e.g., quirks of arithmetic calculation).

It is important to note that these four divisions are the basic structure of the language; that is a very different concept than the way in which the ANSI standard has been constructed. The COBOL 74 Standard is composed of a nucleus and 11 data processing modules. This concept is not particularly important with regard to evaluating COBOL against TINMAN requirements, except that it makes it easy to pinpoint the two modules which are clearly not needed by TINMAN. They are: Sort-Merge and Report Writer. Parts of the other nine modules are also not needed to satisfy TINMAN requirements.

Emphasis on Data and File Handling

The basic purpose of COBOL is to allow efficient handling and minor computation of masses of data which are assumed to be collected on files or mass storage devices. This is what has been classically known as Business Data Processing. Because of the great desire to maintain machine independence, a very important concept in dealing with data is what is known as the

"standard data format". This simply means that from the language point of view and from the user point of view, data will be considered to be in decimal form. This does not require the implementer to store it that way, but allows the user to deal with the data in a machine-independent way.

Naturally, there is a significant orientation towards having effective input/output facilities to deal with the masses of data.

General Approach to Syntax

From its inception, the emphasis in COBOL has been on readability and the desire to make the language as "English-like" as possible. This has been considered an extremely important characteristic of COBOL, as contrasted with a different concept of including and developing an elegant-language structure and elegant-language features. (See related comments in last paragraph of the section below.)

IMPORTANT CHARACTERISTICS RE USAGE AND DEVELOPMENT

Usage and General Views

COBOL is probably the most widely used language, although it is difficult to obtain meaningful statistics on any type and measure of usage. However, the existence of many smaller machines which are used for business data processing tends to support the data (which is not very reliable) but which seems to lead to the conclusion that COBOL is the most widely used language today. It has been implemented on most machines, including very small ones. As is well known, some of the computers that are being called mini-computers in 1976 are more powerful than some of those existing in the early 1960's and COBOL was implemented on many of those early machines. This is important simply because it proves that COBOL can be implemented effectively on almost any machine.

Unfortunately, COBOL is not well thought of by computer scientists or by most language experts. They tend to be disdainful of the English-like nature of the language and its resulting wordiness just as they dislike the inelegant program structure. As in the case of FORTRAN, COBOL's basic mode was set very early (namely in 1959), and while numerous improvements have been made to the language since then, the

framework established at that time has remained fundamentally unchanged.

Development

From COBOL inception in 1959, the language specifications have been under the jurisdiction of a fairly loosely defined organization called CODASYL which has been the standard maintenance body. In addition, under the auspices of the American National Standards Institute, COBOL was first standardized in 1968 and then later a revised standard was developed in 1974. The evaluation with respect to TINMAN in this report is based on the 1974 Standard, with just a few comments on the 1968 version. A newer version (1976) of the language was defined but is not standardized by ANSI.

GENERAL EVALUATION OF COBOL

Strengths

The basic strengths of COBOL considered as a language are the following:

1. Flexible data formats

2. Detailed control of the data formats is available to the programmer, but these details do not have to be specified; some defaults are available

3. Very powerful and flexible data description which could easily be extended to handle new data types

4. Good input/output, particularly allowing for both sequential and random access and mass storage files

5. Readability - COBOL is verbose, but this makes it easily readable to anyone knowledgeable in data processing

6. Probably the most widely used language in existence

today (although this is impossible to either define or measure specifically)

7. Has been widely implemented on most machines, which means that it could probably be implemented on new central processors which might be developed.

Weaknesses

The basic weaknesses of COBOL, considered as a language, are the following:

1. Program structure is very weak. Essentially (only) two levels of procedure hierarchy are allowed, but there is no provision for separate data declarations at different levels.

2. Parameter passage for procedures is very weak.

3. No functions are allowed and this might be hard to add syntactically. (Note that the intended purpose of COBOL is for an area of application where functions are not common nor very necessary.)

4. No extensibility features are in the standard (although they could be added).

5. No pointer mechanisms are available.

SPECIFIC EVALUATION WITH REGARD TO TINMAN

The evaluation of TINMAN with respect to COBOL 68 was done after the evaluation with respect to COBOL 74. Since most implementations of COBOL are of the COBOL 68 Standard, there is some importance associated with an evaluation based on COBOL 68. On the other hand, the long time-span between the first (i.e., 1968) standard, and the more current COBOL 74 version makes the COBOL 74 evaluation far more important in terms of what COBOL actually is today. It is worth pointing out that there is a 'CODASYL COBOL Journal of Development' (1976) which contains still more changes and additions to the COBOL 74 Standard. However, no evaluation of TINMAN was made with respect to the 1976 JOD, and it is believed that it would not significantly change the overall evaluation of COBOL.

In order not to clutter up the main part of the evaluation which is with respect to COBOL 74, the only time that comments are made with respect to COBOL 68 is when the evaluation due to COBOL 68 is lower from that of COBOL 74. If the evaluation remains the same (although the reasoning may differ in the two cases), no comments are made about COBOL 68.

The following are paragraph-by-paragraph comparisons of the TINMAN requirements to the candidate language specifications document:

A1.COBOL.Typed Language
T (Satisfies)

The Data Description (page I-119) in the COBOL DATA DIVISION ensures that this requirement is fully satisfied. For each item of data and for all groupings of data, their types must be fully described in the DATA DIVISION. This is done largely (but not entirely) through the PICTURE clause (pages II-18 through II-26).

A2.COBOL.Data Types
Partially Satisfies

The fundamental concept of describing data in COBOL is different from most other languages. COBOL bases its philosophy (for numeric data types) on the concept of a "standard data format" which is essentially a conceptual decimal representation of numbers. Therefore, integers and fixed-point numbers are easily described simply by indicating the position of the decimal point which is done in the COBOL PICTURE clause (pages II-18 through II-26). There is no floating-point in COBOL. The COBOL standard does not allow Boolean data types, although the "CODASYL Journal of Development" does. However, COBOL allows many types of "conditions" which serve similar purposes, i.e., have values of True or False (see pages II-41 through II-49). Character data type variables are allowed and in COBOL are referred to as alphanumeric (page II-22). Arrays are allowed (although limited to 3 dimensions) by using the OCCURS clause (pages III-2 through III-4). Records are definitely allowed, and in fact form the heart of the concept of organizing data for COBOL programs. They are shown through the use of the level number concept (page II-17).

Floating point exists in at least one compiler and possibly in others; i.e., it is not difficult to do, either from a language or an implementation viewpoint. Since the "CODASYL Journal of Development" includes Boolean data types, the language specifications already exist although this was not included in the standard. From a language point of view, it is trivial to lift the restriction of arrays to 3 dimensions; the implementation techniques for multidimensional arrays are well known although they may not be used in current COBOL compilers.

A3.COBOL.Precision
Partially Satisfies

COBOL does not allow floating-point arithmetic and therefore does not deal with the problem of the precision of floating-point arithmetic calculations. The requirement for precision specification for individual variables is partially met. The precision of each numeric data item is automatically defined as part of the data description of that variable, and in particular by the PICTURE clause (pages II-18 through II-20). In doing arithmetic calculations, a maximum of 18 digits is allowed for computational purposes. It is therefore formally true that the programmer does not have complete control over the precision desired, but on a more practical basis 18 decimal digits would seem to suffice in most cases. The rules associated with the COBOL arithmetic verbs (ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE) all require that the compiler provide enough precision to satisfy the constraints on the precision desired in the result. (See for example ADD, page II-56, point (5).)

For needed changes, note the following: if floating point were added then precision rules could also be included. The limitation on 18 digits could be increased in the language specifications if desired and this would merely require compilers to provide greater precision in their calculations.

A4.COBOL.Fixed-Point Numbers
T (Satisfies)

It is not entirely clear whether COBOL fully or partially satisfies this requirement, since the requirement itself is ambiguous. In general terminology, fixed-point numbers are sometimes interpreted as simply numbers between plus and minus 1, whereas in other cases fixed-point numbers are numbers of the form NNN.NNN with any number of digits on either side of the decimal point. Because of the COBOL philosophy of treating all numbers as decimal, the user specifies the location of the decimal point in the PICTURE clause (pages II-18 through II-20) that will automatically determine the fractional step size. In this context, scale factor management is not really relevant. However, it should be noted that all arithmetic computations are to be implemented essentially by alignment on the decimal point (see page II-51, section 5.3.4). Rounding is controlled by the programmer (pages II-50 and II-55). Integers will, indeed, be treated as exact quantities for the reasons just given.

A5.COBOL.Character Data
T (Satisfies)

The COBOL character set consists of 51 characters, all included within the ASCII Code. The user is allowed to declare a collating sequence as part of the description of the object computer (page II-6). If the collating sequence is specified, it will be used to determine the

truth value of any non-numeric comparisons; if it is not specified, then the collating sequence inherent in the Object computer will be used. The CODE-SET clause (page IV-12) specifies the character code used externally and the algorithm for converting from external to internal usage.

COBOL 68 - Partially satisfies. COBOL 68 does not allow the user to specify the collating sequence.

A6.COBOL.Arrays
T (Satisfies)

COBOL fully satisfies the requirement as written, although COBOL has a number of restrictions which are not prohibited by the requirement. COBOL allows specification of the number of dimensions in the OCCURS clause in the Data Description (pages III-2 through III-4). The range of subscript values are specified in that clause. The type of each array component is specified in the Data Description of the array. The actual maximum subscript value can either be specified at compile time, or can be determined at object time if the programmer uses the DEPENDING ON clause in the OCCURS clause. The use of the OCCURS clause requires the programmer to specify the number of dimensions at compile time. The lowest subscript value must be the number 1, i.e., it cannot be zero or negative (page III-2, section 2.1.3, point (2)). It should be noted that the subscript must be an integer, or a variable which is declared as an integer, i.e., the subscript itself cannot be an expression. Furthermore, subscripts cannot be subscripted (page I-89, section 5.3.3.8.2).

Only three subscripts are allowed (page I-89); however, it would be a trivial language change to permit any number. This requirement is a holdover from the earliest days, and also a reflection of the fact that more than 3 dimensions is not that necessary in normal business data processing problems. The ranges of subscript values are specified in the OCCURS clause. It is worth noting that COBOL also provides a method for handling arrays known as "indexing". This simply means that variables defined as indexes are permitted, but they are not treated as other variables, and are handled in whatever way is deemed most effective for the particular hardware. The programmer has no control over the method, but does determine when index variables are used.

A7.COBOL.Records
T (Satisfies)

The REDEFINES clause in the Data Division (page II-27) permits records to have alternative structures which are defined at compile time. Specifically, "the REDEFINES clause specifies the redefinition of

a storage area, not of the data items occupying the area" (page II-27, section 4.8.4, point 2). There is a restriction that variable length structures cannot be redefined. Discrimination is done by the proper use of the data names, i.e., the programmer uses the names associated with the structure that he wants. There is also a RENAMES clause (page II-29 through II-30) which permits some regrouping of data, but it is not the main data overlay facility; the REDEFINES is.

B1.COBOL.Assignment and Reference
T (Satisfies)

Assignment in COBOL is carried out from the MOVE statement and the five arithmetic statements (ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE). While this assignment facility does permit any value of a given type to be assigned to a variable, the latter must be "large enough" to receive the full value of the "sending data". Since COBOL variables occupy differing amounts of storage (rather than a word or other size predetermined by the compiler), if the "receiving" data item is "too small" to contain the "sending" item then there are rules in the language specification which determine what is to be done. Reference to any data-name does indeed retrieve the last assigned value.

Since COBOL does not permit encapsulated type definitions, the user is not able to define assignment and access operations within them. The wording of the requirement seems to make it legitimate to consider the requirement fully satisfied because it says essentially "If there are encapsulated type definitions then ...."

B2.COBOL.Equivalence
T (Satisfies)

The "IF condition" statement in COBOL provides for logical identity tests. Numeric operands are compared on the basis of algebraic value, regardless of internal format (page II-42, section 5.2.1.1.1). Non-numeric operands are compared on a character by character basis (pages II-42 through II-43). It is allowable to compare an integer variable or literal to a non-numeric operand (page II-42, section 5.2.1.1.2). This is done by essentially defining the integer as an alphanumeric variable and then comparing two non-numeric operands. This would appear to violate the requirement that elements of disjoint types never be identical, but it seems legitimate to consider this within the "10% deviation" allowed for a "fully satisfies" determination.

For needed changes, note the following: the exception to 100% compliance noted above (i.e., ability to compare numeric and non-numeric operands) could easily be removed from the language.

B3.COBOL.Relationals
T (Satisfies)

The "fully satisfies" evaluation refers to the requirement actually stated in B3 that "Relational operations will be automatically defined for numeric data". It also satisfies the other part of the requirement by default because COBOL does not have types defined by enumeration. As for the requirements stated in the text, COBOL contains the 3 relational operators "greater than", "less than", "equal to" and also has the "not" operator which then permits the same result as having "greater than or equal to", "less than or equal to" and "unequal". As far as inhibiting ordering definitions, there are no types defined by enumeration in COBOL; thus there is no need to be concerned with unordered sets.

B4.COBOL.Arithmetic Operations
T (Satisfies)

See ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE verbs for the basic computation. Division with and without remainder is permitted (page II-61, Format 4). Exponentiation and negation operators exist (page II-39). It should be noted that there are no floating-point variables and hence no arithmetic operations to be applied to them.

For needed changes, note the following: if floating point was to be added as a data type, no change or addition to the syntax of the existing 5 arithmetic verbs is needed.

B5.COBOL.Truncation and Rounding
T (Satisfies)

In the most literal sense of the words, the COBOL specifications violate the truncation rule; however, within any reasonable interpretation of what is intended by the requirement, COBOL satisfies it completely. To be more specific, the whole concept of arithmetic and assign operations on data in COBOL is based on the concept of aligning on the decimal point. Arithmetic is carried out on numbers after alignment on the decimal point. More importantly, when results are assigned (by any of the 5 arithmetic verbs or the MOVE statement) the result is based on the described format of the receiving field based primarily on alignment of the decimal point. Thus, if a programmer chose to store an integer which had 10 digits in it into a field which had only 5 digits in it, then there would indeed be truncation of the 5 most significant digits. However, this is completely under programmer control and the compiler has nothing to do with it whatsoever. It is being assumed that the reasonable interpretation of the requirement is that the compiler cannot perform a truncation of the most significant digits of the numeric quantity of its own volition. Rounding occurs on

the least significant digits and is carried out only under explicit programmer control with the use of the ROUNDED option in the arithmetic verbs.

For needed changes, note the following: there are no floating-point numbers in COBOL, but if they were added these requirements could certainly be satisfied.

B6.COBOL.Boolean Operation
Partially Satisfies

The operations, "and", "or", "not" are supplied (page II-45). There is no "exclusive or" in the standard although it does exist in the "CODASYL Journal of Development" (page III-7 through III-10). It should be noted that although there are no Boolean data types as such, the 3 indicated operators are used between "conditions" which are essentially expressions having a truth value (pages II-41 through II-49). There is no specification in the language about the methodology for evaluating "and" and "or", i.e., the language does not contain a requirement for evaluation in short circuit mode. This is the only reason that COBOL does not fully satisfy this requirement. Note that the requirement asks for a NOR capability but at a 10 November 1976 meeting at IDA, it was agreed that XOR was intended.

For needed changes, note the following: an exclusive or (XOR) can very simply be added, for it already exists in the "CODASYL Journal of Development" (page III-7 through III-10).

A capability to provide short circuit evaluation could also be added very simply.

B7.COBOL.Scalar Operations
T (Satisfies)

Without specifying all of the major details, the COBOL MOVE verb (pages II-74 through II-76) does permit the required transfers. It should be noted that the correspondence will be by position unless the programmer specifies MOVE CORRESPONDING in which case it will be done by alignment of data names.

B8.COBOL.Type Conversion
Fails to Satisfy

The reason that COBOL fails is because the fundamental principle

behind doing calculations on numeric data is alignment on the decimal point independently of the internal representation. Therefore, conversion operations between integers and fixed-point numbers are not specified explicitly by the programmer. Furthermore, the facility to show different "usages" for arithmetic (which refer to internal representations specified by the implementer) means that there are even more implicit conversions! In addition, because of the richness of data types in COBOL (involving alphabetic, alphanumeric, alphanumeric edited, numeric and numeric edited) as described in the PICTURE clause (pages II-18 through II-26) there are numerous rules spelled out as to what conversions are legal and how those are to be carried out automatically in performing computations and/or assignments. These rules are explicitly stated in the language specification and are not at the discretion of the implementer. Furthermore, the programmer is in complete control because any conversions carried out are based on the way he described the data. If "implicit type conversions" means that the data description controls the conversion, then COBOL conversions are all implicit.

For needed changes, note the following: it would not be possible to modify COBOL to meet this requirement because COBOL's current way of handling these conversions is a fundamental part of the COBOL concept.

B9.COBOL.Changes in Numeric Representation
Partially Satisfies

Certainly COBOL does not require explicit conversion operations between numerical ranges. However, the requirement that "There will be a run time exception condition when any integer or fixed-point value is truncated." is ambiguous. It is not clear whether the requirement means that there should be an automatic transfer to some predetermined routine whenever this error occurs, or whether it is meant to say that the programmer can specify what is to happen if this error occurs. The "ON SIZE ERROR" clause in the 5 arithmetic verbs permits the programmer to determine what is to happen if truncation or other errors such as division by zero occur.

For needed changes, note the following: either interpretation would be easy to include in COBOL.

B10.COBOL.Input/Output Operations
T (Satisfies)

Part of COBOL's major strength lies in its powerful I/O capability. The Sequential I/O module permits programmer interaction with sequential files, independent of the particular physical device involved. The connection between the program files and the hardware is made by the

user through what he writes in his program in the ENVIRONMENT DIVISION and specifically in the File Control paragraph of the Input/Output section (pages IV-4 through IV-5). The verbs READ and WRITE permit input and output of logical records respectively. The ACCEPT and DISPLAY verbs are used for low-volume data input and output respectively and, in particular, can be used effectively for terminals. In addition to this, there is a Relative I/O module which provides the capability of accessing records of a mass storage file in either a random or sequential manner.

There are a number of ways in which the user has control over exception conditions. In particular, the AT END condition used in the READ (page IV-28) and WRITE (page IV-34) verbs allows the user to specify what is to be done when the end of file is reached. There is a USE statement (pages IV-32 through IV-33) which permits the user to specify procedures for the input/output error handling that are in addition to the standard procedures provided by the input/output control system. Specifically, the user can specify the program steps to be taken upon occurrence of an error or an exception. This is not installation-dependent; however, the user is able to specify in the ENVIRONMENT DIVISION the relation between the logical files and the physical hardware that is involved. Furthermore, the I/O-CONTROL paragraph (pages IV-6 through IV-8) permits the user to specify how reruns are to be handled.

The only aspect of this which COBOL seems to fail to meet 100% is the ability to dynamically assign and reassign devices; i.e., there is no way the programmer can change the assignment of physical devices at execution time.

For needed changes, note the following: the language could be modified to permit dynamic allocation of devices, but it would probably require major changes in the concept of the ENVIRONMENT DIVISION and would certainly increase implementation difficulties.

B11.COBOL.Power Set Operations
Fails to Satisfy

COBOL does not allow power sets of data types, nor enumeration types.

For needed changes, note the following: it would be moderately difficult to add these facilities.

C1.COBOL.Side Effects
T (Satisfies)

The COBOL standard states with respect to arithmetic expressions that "when the sequence of execution is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right" (page II-40, point 2).

C2.COBOL.Operand Structure
T (Satisfies)

The operator hierarchies involved pertain to (1) the arithmetic operations, (2) the logical operations and (3) the relationship between them. There are the obvious 4 levels for arithmetic operation, the obvious 2 levels for logical operation (pages II-39 and II-49, respectively). However, there are several types of conditions (which are really Boolean expressions although called "conditions" in COBOL) which also have a hierarchy (page II-48). Parentheses are of course allowed in all the appropriate places, either to overcome the normal hierarchy, or to specify an execution order within elements of the same precedence level. The condition evaluation rules, when both arithmetic and conditional expressions are involved, are well defined (pages II-47 through II-49). The user is not able to define new operator precedence rules, nor to change the precedence of existing operators.

C3.COBOL.Expressions Permitted
Fails to Satisfy

COBOL has a great many restrictions on the use of expressions. For example, subscripts cannot be expressions, they cannot be subscripted, etc.

For needed changes, note the following: it would be difficult, and perhaps impossible, to modify COBOL to meet this requirement.

C4.COBOL.Constant Expressions
Fails to Satisfy

COBOL allows literals (i.e., constants) in many places but does not permit constant expressions where constants are allowed.

For needed changes, note the following: it probably would be difficult to add this facility because it might make the syntax ambiguous. It would certainly make the parsing more difficult.

C5.COBOL.Consistent Parameter Rules
Partially Satisfies

Since COBOL does not have many instances of parameters, there are not too many rules. There is only one way to invoke a procedure, and this is defined in the Inter-Program Communication module (pages XII-1 through XII-8). The parameters are defined as part of the PROCEDURE DIVISION heading (page XII-4). As far as exception handling is concerned, there is consistency throughout those in the sense that whenever the specific clauses (e.g., ON OVERFLOW, ON SIZE ERROR) are used, each is followed by an imperative statement. Whether this can, or should be, considered a parameter is certainly debatable. Thus, to some extent COBOL satisfies this requirement almost by default.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: parameter handling is very weak in COBOL. It would require significant changes to improve it.

C6.COBOL.Type Agreement in Parameters
Fails to Satisfy

While the most accurate judgment would seem to be "fails", it should be pointed out that this is not 100% certain. There appears to be an error in the text of the COBOL standard. The method for handling subroutines in COBOL (Section XII) specifies that the subroutine will contain a Linkage Section which gives the data descriptions for the formal parameters (page XII-2). The specifications say "Within a called program, Linkage Section data items are processed according to their data descriptions given in the called program" (page XII-4). The specification then goes on to refer to the actual parameters, and then says "Their descriptions must define an equal number of character positions; however, they need not be the same name". If the word "name" was not there, the rule would be very explicit that the formal and actual parameters need not be of the same type (which would then violate the TINMAN requirement). However, the wording in the "CODASYL Journal of Development" says "Except that they must define an equal number of character positions, their descriptions need not be the same" (page III-7, point 2). In COBOL terminology, if data items do not have the same description, they are not necessarily of the same type. Thus the COBOL standard seems ambiguous whereas the "CODASYL Journal of Development" is quite clear (and violates the TINMAN requirement). It is not clear from the specifications what happens to data arrays.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: it would be fairly easy to change COBOL to meet this requirement.

C7.COBOL.Formal Parameters
Fails to Satisfy

COBOL does not provide a special class of data parameter to serve as a constant. It does provide the second class of data parameter, namely the variable, and the passage of parameters is a CALL by location (page XII-2). There are no formal parameter classes for control action on exception condition nor for procedure parameters. However, as described under B10, COBOL does have very good language facilities for handling exceptions.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: it would be quite difficult to modify COBOL to meet this requirement. It would require fundamental changes in concepts.

C8.COBOL.Formal Parameter Specifications
T (Satisfies)

COBOL requires a Linkage Section in the DATA DIVISION of the procedure declaration and this Linkage Section describes the data to be used in the procedure. This must be available at compile time. However, the specifications state "Within a called program, Linkage Section data items are processed according to their data descriptions given in the called program" (page XII-4). Thus the "specification of ... parameters" in the procedure declaration is essentially overridden by the calling program.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

C9.COBOL.Variable Numbers of Parameters
Fails to Satisfy

The actual names of all the parameters must be shown at compile time.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: it would be fairly easy to

add this to the language specifications, but the implementation would have the normal difficulties associated with it.

D1.COBOL.Constant Value Identifiers
T (Satisfies)

There is a facility called the VALUE IS clause (pages II-36 through II-38) which is part of the Data Description and which allows the assignment of specific literals (i.e., constants) to data at compile time. This seems to satisfy the spirit of this requirement. Constants can also be assigned to data items in the Communications Section and to a condition name (I-78 and II-37, section 4.13.5) and to items in the Report Writer Section. Although section and paragraph names cannot have constants assigned to them, that capability doesn't seem to make sense anyway within the overall spirit of TINMAN's avoidance of "tricky" coding.

D2.COBOL.Numeric Literals
T (Satisfies)

COBOL provides for both numeric and non-numeric literals (pages I-80 through I-82). Numeric literals are limited to 18 digits and can contain (only) one decimal point. The internal representation of this literal will be consistent with the internal representation of all other data given in the standard data format of COBOL. COBOL also allows for non-numeric literals of up to 120 characters in length. There is currently no rule in COBOL that says "numeric constants will have the same value in both programs and data" but it would be a trivial rule to impose.

The requirement that "numeric constants will have the same value .. in both programs and data..." is a directive to the implementer and not a language specification. There is no such statement in the COBOL specifications but it would be trivial to include it.

D3.COBOL.Initial Values of Variables
T (Satisfies)

The VALUE IS clause (pages II-36 through II-38) enables the user to specify the initial values of variables as part of their declaration, but does not require him to do so. There are no default initial values in COBOL. Because of the fundamental way in which COBOL approaches data it is meaningless to have a special provision "for run time testing for initialization".

D4.COBOL.Numeric Range and Step Size
T (Satisfies)

The PICTURE CLAUSE (pages II-18 through II-19) for numeric items allows (and requires) the user to specify the number of decimal digits involved.

The number of occurrences of the digit "9" combined with the presence of an actual or assumed decimal point indicate the maximum number of digits and hence the range of the numeric variables. As an example, a "picture" of 99999V9999 would represent a number that has four decimal places and a whole value of less than 100,000; the V represents the location of an assumed decimal point. If a period is used, it is treated as an actual decimal point; thus replacing V by "." represents numbers of the same range. The step size is automatically the lowest digit position, i.e., the digit position furthest to the right of the decimal point. (In the above case, the step size is .0001.) If the assumed decimal point is actually outside the number of significant digits (represented by the string of 9's) the user specifies this by using a string of P's on the right or left as appropriate. Thus 99 PPP represents numbers from 00000 to less than 100000 with only 2 significant digits. Numbers cannot be more than 18 decimal digits long.

D5.COBOL.Variables Types
T (Satisfies)

Arrays can be components of records via the OCCURS clause (page III-2) and records can be components of arrays (page III-2). Data items can be given any range of values.

D6.COBOL.Pointer Variables
Fails to Satisfy

COBOL does not have any pointer variables.

For needed changes, note the following: it would be difficult, if not impossible, to add a pointer mechanism to COBOL.

E1.COBOL.User Definitions Possible
Partially Satisfies

This evaluation is valid if and only if subroutines are considered a form of defining new operations. If not, then COBOL fails completely on this requirement.

For needed changes, note the following: the ability to add new syntactic operations (i.e., new verbs and not just subroutines) is conceptually very easy; in fact, from its earliest published specification (1960) until 1968, COBOL had a DEFINE verb which enabled the user to create a new verb with specified syntax which could then be used just as any other verb. DEFINE was eventually dropped from the language specification primarily because nobody had implemented it. There are some commercial software houses which have developed add-on packages to COBOL which permit the equivalent of this and so it can be implemented.

Defining new data types in terms of the base language would probably not be very difficult because of the existing power and complexity of the existing COBOL data types.

E2.COBOL.Consistent Use of Types
Fails to Satisfy

COBOL does not enable the user to define new data types.

For needed changes, note the following: defining new data types in terms of the base language probably would not be very difficult because of the existing power and complexity of the existing COBOL data types.

E3.COBOL.No Default Declarations
Fails to Satisfy

COBOL does have default declarations. At least the following have been noted: if the USAGE clause is not specified, the usage is implicitly DISPLAY (page II-35, point (5)). In the SYNCHRONIZE clause, if neither LEFT nor RIGHT is specified, then it is assumed that the item will be positioned in an efficient manner as determined by the implementer (page II-33, point (2)); in the SIGN clause, if the SEPARATE CHARACTER phrase is not present, then certain other assumptions are made (pages II-31 through II-32, point (3)). There may be other defaults as well.

For needed changes, note the following: the elimination of all these defaults is very simple; one merely specifies exactly what is to happen in each existing default case and requires the appropriate item to be put into the language specification. An alternative way of handling it is to remove the optional status of these clauses and include another alternative to those already existing.

E4.COBOL.Can Extend Existing Operators

Fails to Satisfy

COBOL fails by default to satisfy this requirement because it does not allow the creation of new data types.

For needed changes, note the following: defining new data types in terms of the base language probably would not be very difficult because of the power and complexity of the existing COBOL data types.

E5.COBOL.Type Definitions
Fails to Satisfy

COBOL fails by default because it does not allow type definitions.

For needed changes, note the following: this probably would not be very difficult to include in COBOL because of the existing power and complexity of COBOL data types. However, even if this requirement could be met, it would then cause difficulty in complying with the other Definition Facility requirement (i.e., category E).

E6.COBOL.Data Defining Mechanisms
Fails to Satisfy

COBOL fails by default because it does not permit the user to define new types.

For needed changes, note the following: this probably would not be very difficult to include in COBOL because of the existing power and complexity of COBOL data types. However, even if this requirement could be met, it would then cause difficulty in complying with the other Definition Facility requirement (i.e., category E).

E7.COBOL.No Free Union or Subset Types
T (Satisfies)

COBOL fully satisfies this by default, i.e., it does not permit type definitions at all and hence does not permit them in a way that is deemed undesirable.

E8.COBOL.Type Initialization
Fails to Satisfy

COBOL fails by default because it does not permit the user to define a new type.

For needed changes, note the following: this probably would not be very difficult to include in COBOL because of the existing power and complexity of COBOL data types. However, even if this requirement could be met, it would then cause difficulty in complying with the other Definition Facility requirement (i.e., category E).

F1.COBOL.Separate Allocation and Access Allowed
Fails to Satisfy

This requirement is really meaningful only in a language which has "own variables" and a true block structure. Since COBOL does not have that type of program structure, this requirement would not be needed in COBOL and hence is not there. Some control over storage and overlay are provided by the Segmentation facility.

For needed changes, note the following: it would be neither possible nor sensible to try to put this into COBOL.

F2.COBOL.Limiting Access Scope
Partially Satisfies

COBOL does not allow type definitions so problems of access in that connection do not arise. It is possible to associate new local names with separately defined programs. This is done via the Linkage Section of the Inter-Program Communication Module (page XII-4).

For needed changes, note the following: it would be very difficult to add the missing features to COBOL.

F3.COBOL.Compile Time Scope Determination
T (Satisfies)

The term "scope" is not used in COBOL and is not really applicable to COBOL as it is to the block-structured languages such as ALGOL and PL/I. However, COBOL definitely satisfies the requirement that "the scope of identifiers will be wholly determined at compile time". This is achieved through the rules on qualification for both data names and procedure names (pages I-87 through I-89).

F4.COBOL.Libraries Available

T (Satisfies)

COBOL permits the handling and the use of libraries through the Inter-Program Communication module, which allows procedure definitions for routines which are to be separately compiled, and allows a CALL statement in the basic program. Both procedures and data descriptions can be obtained from a library and included directly in a source program via the COPY statement in the Library Module (pages X-2 through X-4). The development and availability of libraries is of course beyond the scope of a language evaluation; however, nothing in COBOL prevents the development of such libraries.

COBOL 68 - Partially satisfies. Only one library can be accessed from within a single program, and there is no provision for subroutines with parameters.

F5.COBOL.Library Contents
T (Satisfies)

COBOL allows the existence of Libraries through both the Library and the Inter-Program Communication module. In addition, the ENTER verb (page II-63) provides a means of allowing the use of languages other than COBOL.

F6.COBOL.Libraries and Compools Indistinguishable
T (Satisfies)

COBOL contains a COPY statement (page X-2). This COPY statement can be used for both programs and data descriptions, and can be used almost anywhere in the source COBOL program. The specifications state "COBOL libraries contain library texts that are available to the compiler for copying at compile time. The effect of the interpretation of the COPY statement is to insert text into the source program, where it will be treated by the compiler as part of the source program" (page X-1). This applies to both procedures and data descriptions as well as other elements of a COBOL program.

F7.COBOL.Standard Library Definitions
T (Satisfies)

In the CONFIGURATION Section of the ENVIRONMENT DIVISION there is a standard format for describing the source computer, the object computer, various hardware switches, etc. (page I-113). COBOL does contain a standard way of accessing particular pieces of hardware such as a real-time clock through the ACCEPT verb. In the FILE CONTROL paragraph,

the user is allowed to specify a file RERUN EVERY integer CLOCK UNITS. Furthermore, the Communication module provides standard ways of dealing with message control systems (page XIII-3).

COBOL 68 - Partially satisfies. There is no provision in the ACCEPT verb of COBOL 68 for obtaining information from the real-time clock. Furthermore, COBOL 68 does not contain the Communication module and hence no way of dealing with message control systems.

G1.COBOL.Kinds of Control Structures
Partially Satisfies

COBOL has normal sequential control through the executable statements. The conditional facility is handled by the IF statement (pages II-66 through II-67), as well as by transfers of control which occur when exceptions are handled. Iterative control is provided by the PERFORM statement (pages II-78 through II-84). COBOL does not allow recursive routines. This is implied as part of the CALL statement specifications which say "On all other entries into the called program, the state of the program remains unchanged from its state when last exited. This includes all data fields, the status and positioning of all files, and all alterable switch setting." (page XII-5, point 3). (The "all other entries" refers to the initial call or after a CANCEL verb is executed.) COBOL does not provide control structures for parallel processing, except that the input/output operations may be implemented for parallel execution without any further action by the user. COBOL does provide a great deal of exception handling. In particular, COBOL allows the programmer to specify what will happen in the case of a size error in arithmetic computation, an overflow of memory in calling a subroutine, end of file, and several more spcialized cases. (Partial list is on page I-102.) In addition, the USE statement (pages IV-32 through IV-33) allows the user to specify procedures for input/output error handling in addition to the standard ones provided by the system.

For needed changes, note the following: to permit recursion in COBOL is no more difficult than in any other language as long as one is willing to pay the penalty at object time. It would be very easy to have the language require that a subroutine be declared recursive as part of its heading, and this would permit the implementer to avoid including the object time package to deal with non-existent recursion. The parallel processing could probably be added relatively easily.

G2.COBOL.The GOTO
Partially Satisfies

COBOL contains the simple GOTO statement (page II-65). Since there

are no scope rules in the sense of the block-structured languages, the GOTO can be used to transfer control to labels anywhere in the program. However, there is an ALTER statement (page II-57) which allows the change of the label in the "GOTO label" at object time. This unfortunately "encourages its use in dangerous and confused ways".

For needed changes, note the following: it would be trivial to bring COBOL into full compliance with this requirement by eliminating the ALTER statement. In fact, the "CODASYL Journal of Development" for 1976 has removed ALTER from the language.

G3.COBOL.Conditional Control
Partially Satisfies

The basic conditional control statement in COBOL is the IF (pages II-66 through II-67). This partially violates the requirement that the language specify that an ELSE clause appear after each IF...THEN, because under certain conditions the ELSE can be eliminated - namely when control is to go to the next sentence. The "condition" following the IF is extremely general (pages II-41 through II-46). The user is allowed to make fairly standard relational tests of size among arithmetic expressions, but can also (1) test the class conditions (i.e., whether the operand is numeric or alphabetic), (2) test a condition name (which specifies specific values which can be tested against), (3) test switch status (refers to the on or off status of an implementer-defined switch and (4) use a separate sign test to determine whether an arithmetic expression is positive, negative or zero.

For needed changes, note the following: it would obviously be a trivial language change to require the existence of an "ELSE" after each "IF...THEN".

G4.COBOL.Iterative Control
T (Satisfies)

The PERFORM verb is the basic iterative control statement. It does permit the termination condition to appear anywhere in the loop. The PERFORM statement allows iteration on three nested variables with a different termination condition for each one all in one PERFORM statement. Entry is allowed only at the head of the loop. The second format of the PERFORM statement allows the user to specify a fixed number of iterations. The language specifications do clearly indicate the value of a control variable after termination (see pages II-82 through II-83).

G5.COBOL.Recursive Routines

Fails to Satisfy

COBOL does not allow recursive routines. This is stated as part of the CALL statement specifications which say (page XII-5, point 3) "On all other entries into the called program, the state of the program remains unchanged from its state when last exited. This includes all data fields, the status and positioning of all files, and all alterable switch settings." (The "all other entries" refers to the initial call or after a CANCEL verb is executed.)

For needed changes, note the following: to permit recursion in COBOL is no more difficult than in any other language as long as one is willing to pay the penalty at object time. It would be very easy to have the language require that a subroutine be declared recursive as part of its heading, and this would permit the implementer to avoid including the object time package to deal with non-existent recursion.

G6.COBOL.Parallel Processing
Fails to Satisfy

COBOL does not contain any user-controlled parallel processing capability.

For needed changes, note the following: it is probably not too difficult to add parallel processing to COBOL.

G7.COBOL.Exception Handling
T (Satisfies)

COBOL has very good exception handling. A list of exception cases that the user can control is contained in the "Conditional Statement" list on page I-103. In addition, there is a USE verb which permits special handling of files beyond the normal system defined facilities. In each of these "exception cases", the user specifies the statement that is to be executed in the exception situation. This statement can be either a GOTO, or a call to a library routine, or a single statement.

G8.COBOL.Synchronization and Real-Time
Partially Satisfies

COBOL only partially satisfies this requirement because it does not have the fundamental capability of handling parallel processing. The ACCEPT verb allows the user access to Date, Day and Time from the hardware. There is a Communication module which provides the user with

considerable facility for the specialized application of teleprocessing (see pages XIII-1 through XIII-23). In this facility it is assumed that there is a Message Control System (MCS) beyond the reach of the programmer. The purpose of this particular facility is to provide the interface between the source program and the user. The system allows the user to receive and send messages to enable and disable the appropriate input/output devices.

For needed changes, note the following: considering the overall structure of COBOL and the ability of the user to make certain specifications in the ENVIRONMENT DIVISION, it would probably be quite possible and feasible to add this facility without disturbing the overall structure of COBOL from a syntactic point of view.

H1.COBOL.General Characteristics
Partially Satisfies

COBOL is certainly the most readable of all programming languages. It satisfies many, but not quite all, of the specific requirements of H1. Dealing with each one in turn, the following comments should be made. The source language is partially free format. It is only partial because there is a Reference Format in which certain subunits of the program must start in certain columns. Aside from that, the programmer may write in any form that he chooses. (The full details of the Reference Format are found on pages I-105 through I-108.) COBOL does not have a specific statement delimiter, but it does have a sentence delimiter. A sentence is composed of sequential statements. Thus, one determines the beginning of a new statement by the fact that it starts with the appropriate key word, normally a verb. The distinction between a statement and a sentence is important in certain flow-of-control situations. However the user can write each statement as a separate sentence if he chooses in which case this part of the requirement is met.

COBOL certainly allows mnemonically significant identifiers since they are allowed to be up to 30 characters with imbedded hyphens allowed for easy readability.

There are a few abbreviations of key words allowed, but they are specified in the syntax itself. For example, CORRESPONDING can be abbreviated as CORR, but unless the syntax specifically shows a shortened version of a long word (which makes the short version a reserved word) no abbreviations are allowed. COBOL is syntactically unambiguous.

For needed changes, note the following: it would be difficult - and perhaps impossible - to make COBOL completely free format.

H2.COBOL.No Syntax Extensions
T (Satisfies)

COBOL does not permit the user to create any syntax extensions.

H3.COBOL.Source Character Set
T (Satisfies)

COBOL uses a 51-character set (see page I-54). All of these characters are contained within the 64 character ASCII subset.

H4.COBOL.Identifiers and Literals
Partially Satisfies

COBOL allows identifiers of up to 30 characters. They are composed of letters, digits, and the hyphen. The hyphen can be used anywhere except at the beginning and the end; thus the hyphen serves as a break character (see pages I-76 through I-77). Literals are permitted and are of two types - numeric and non-numeric (pages I-80 through I-81). The numeric literals are limited to 18 digits in length which is consistent with the rules involving arithmetic in COBOL. Non-numeric literals are limited to 120 characters in length. There is no break character used internally for literals; thus, a hyphen or space would be considered as its own specific character within a literal. There is no requirement for separate quotation around each line of a long literal; thus literals can be continued on a second line.

For needed changes, note the following: it would be trivial to impose the rule of requiring literals not to exceed one line, i.e., require separate quoting of each line. However, it would be extremely difficult to impose a rule requiring or permitting a break character within a literal because there is no way then to use that break character as a specific character itself. The common break characters of space and hyphen already have very specific syntactic roles in COBOL and changes in those rules would change much of the syntactic structure.

H5.COBOL.Lexical Units and Lines
Fails to Satisfy

COBOL does not contain a provision prohibiting the continuation of lexical units cross lines. It specifically allows this split of lexical units between lines (see the Reference Format discussion on page I-106, section 5.8.2.2.).

For needed changes, note the following: it would be trivial to include such a rule in COBOL.

H6.COBOL.Key Words
Partially Satisfies

The COBOL key words are reserved; i.e., they cannot be used by the user nor by the implementer (see page I-79, section 5.3.2.2.1.3). Key words are not usable in contexts where an identifier can be used. While COBOL has a very large number of reserved words by actual count - specifically about 300 (see page I-109 through I-110); this is really not nearly as bad as it might seem on the surface. Each of these words appears in very specific places in a very logical and readable way in the formats, but there is not likely to be any difficulty to the user in avoiding these unless he wants to go out of his way to make life difficult. The key words are extremely informative and quite specific in their intent and content. It should be pointed out that in COBOL there are actually three significant categories of words - reserved words which cannot be used by anybody; system names which are defined by the implementer; and user-defined words which cannot be either of the other two classes. There is no problem on selection or portability with the reserved words, since they are clearly listed; on the other hand, since the system name words are defined by the implementer, this could conceivably cause difficulty in portability and compatibility (see pages I-76 and I-78).

For needed changes, note the following: it would be absolutely impossible to reduce the reserved word list to a small number without simultaneously destroying the structure and psychology of COBOL.

H7.COBOL.Comment Conventions
Partially Satisfies

Comments in COBOL are defined by having an asterisk in the continuation indicator portion of a line (page I-108). This prevents the comment from being interspersed in the middle of other portions of the program; i.e., each comment must start on a new line. This of course provides automatic termination of the end of the line but, by putting an asterisk in the next line, comments can run over as many lines as desirable. This does not prohibit automatic reformatting of programs.

For needed changes, note the following: it probably would be difficult to allow comments interspersed within the program other than as a separate line.

H8.COBOL.Unmatched Parentheses
T (Satisfies)

COBOL does not permit unmatched parentheses in any form.

H9.COBOL.Uniform Referent Notation
T (Satisfies)

COBOL satisfies this by default because COBOL does not allow function calls.

H10.COBOL.Consistency of Meaning
T (Satisfies)

The only place in which COBOL fails this requirement at all is the use of the "=" in the COMPUTE verb (page II-58). COBOL allows the user to write "COMPUTE identifier = arithmetic expression". This is a form of assignment statement. Since assignment statements rarely occur in the COMPUTE verb, but are derived from the use of the four individual arithmetic verbs, or the MOVE verb, this is not very significant. The "=" is used for a relational operator except for this one case.

For needed changes, note the following: it would be trivial to change the COMPUTE verb format, and replace the "=" by a key word, e.g., AS.

I1.COBOL.No Defaults in Program Logic
T (Satisfies)

There are no defaults which the implementer can specify and which affect program logic. There are implicit transfers of control (e.g., from the PERFORM verb or from a CALL) but these are all clearly defined in the language specifications.

I2.COBOL.Object Representation Specifications Optional
T (Satisfies)

It is actually quite difficult to determine whether any language satisfies this requirement unless the language designers describe all the defaults in one place and in some detail. However COBOL does provide many object representation defaults which are useful. For example, in the SYNCHRONIZED clause of the Data Description, (pages II-33 through II-34) if neither RIGHT nor LEFT is specifed, then the

implementer determines the best positioning of the data item. As another example, in the READ verb (page IV-28) the user has a choice of specifying the name of the area into which the records are to be put or merely leaving them essentially in an input buffer. Furthermore, the matter of whether the "next record" is physically moved or whether an internal pointer mechanism is used is entirely in the hands of the implementer, as are all details of I/O handling.

I3.COBOL.Compile Time Variables
T (Satisfies)

COBOL probably satisfies this requirement better than any other language through its ENVIRONMENT DIVISION. Specifically, the OBJECT COMPUTER paragraph (pages II-6 through II-7) allows the user to specify not only the name of the computer but its memory size, and its collating sequence. In addition, the SPECIAL NAMES paragraph (pages II-8 through II-10) allows the user to specify the character set to be used as well as various types of switches which can be tested during the course of the execution of the program. The INPUT-OUTPUT Section (pages IV-4 through IV-5) in the ENVIRONMENT DIVISION allows the user to specify a great deal about the files and their relationship to the actual hardware and internal buffering at object time. Compile time variables are dealt with through the concept of condition names and the ability to test the "on" or "off" status of an implementer-defined switch (page I-78, section 5.3.2.2.1.1.1 and page II-44, section 5.2.1.4).

I4.COBOL.Conditional Compilation
Fails to Satisfy

COBOL does not provide this capability in any general way. There is a particular case - namely the Debug Module (Section XI) - which does provide a form of compile time conditional statement.

For needed changes, note the following: it would be easy to include this once the requirement was more fully specified.

I5.COBOL.Simple Base Language
Fails to Satisfy

COBOL is not a simple language. It is powerful and hence complex. It does contain duplication of features in order to satisfy differing approaches to how those features should be made available.

It should be noted that this requirement seems incompatible with

the others; to satisfy all the other TINMAN requirements will require a powerful - and not a simple - language.

For needed changes, note the following: there is no way that COBOL could be changed to satisfy this requirement.

I6.COBOL.Translator Restrictions
Fails to Satisfy

COBOL does not supply these limits except in a few very special cases. There are a number of items defined by the implementer, and in fact, a list is given on pages I-7 through I-8. However, most of these items apply to naming of certain things such as the computer name in the ENVIRONMENT DIVISION.

For needed changes, note the following: it would not be any more difficult to specify these restrictions for COBOL than for any other language.

I7.COBOL.Object Machine Restrictions
T (Satisfies)

COBOL does not contain language restrictions which are dependent on the object environment. It is not possible to comment on this for COBOL translators in general.

J1.COBOL.Efficient Object Code
Partially Satisfies

COBOL has an enormous number of optional features which the programmer can include or exclude in his source program. Presumably any good translator will not produce object code for these optional features when they are not used in the program. The input/output specifications assume that the interface with the hardware is being handled outside the translator. There is no automatic movement of programs between main and backing store except under programmer control via the Segmentation Module (section IX). The Segmentation Language Facility allows the user to indicate what is to be fixed in the memory at object time, and what can be overlaid.

For needed changes, note the following: we cannot comment on the required modifications without detailed study of many translators.

J2.COBOL.Optimizations Do Not Change Program Effect
Unknown

This is a requirement on translators and cannot be answered from the language specification.

J3.COBOL.Machine Language Insertions
Partially Satisfies

COBOL's access to machine-dependent hardware facilities is through ENVIRONMENT DIVISION, and so that portion of the requirement is met.

COBOL encapsulates the access and appearance of machine language code insertions through the use of the ENTER verb (page II-63). The machine language statements can either be written "in line" after the ENTER verb or elsewhere, but will be executed as if they had been compiled immediately following the ENTER statement. So COBOL satisfies the basic encapsulation requirement. It is unclear whether the requirement of "not be allowed interspersed with executable statement of the source language" is satisfied by this, but the most reasonable/likely interpretation would lead to the conclusion that COBOL does satisfy it. However, COBOL certainly fails the requirement that these insertions be permitted only within the body of the compile time conditional statements since there are no compile time conditional statements.

For needed changes, note the following: there would be no difficulty in satisfying the part of this requirement that involves inclusion only in compile time conditional statements if they existed (per requirement I4). However, this part of requirement J3 seems to defeat the purpose of allowing machine language instructions in the source program at all.

J4.COBOL.Object Representation Specifications
T (Satisfies)

The Data Description (page I-119) permits the user to specify the object representation of data. The order of fields is shown by the simple sequencing and level structure of COBOL (page II-17), the width of fields is shown via the PICTURE clause (pages II-18 through II-26); the presence of "don't care" fields is shown by using the word FILLER (page II-15) and the position of word boundaries can be controlled by the SYNCHRONIZED clause (pages II-33 through II-34) The user can also specify JUSTification of non-numeric data items (page II-16). However, it is not possible within COBOL to associate source language identifiers

with special machine addresses (to do so certainly would limit portability).

In connection with the text requirement that "object data specifications should be used sparingly", it is important to note that since COBOL has as its objective the efficient handling of large masses of data and data files, it is expected that the user will include the object data information in all cases. However, that would not be necessary for cases in which the data representation was not very important. It seems impossible to judge what language features are "spartan" and what ones are "grandiose".

The primary user control for positioning of data with respect to word boundaries is the SYNCHRONIZED clause (pages II-33 through II-34). If the user does not specify SYNCHRONIZE LEFT or RIGHT, then the translator will determine the most effective positioning of the data item.

J5.COBOL.Open and Closed Routine Calls
Fails to Satisfy

COBOL does not provide this capability in any form.

For needed changes, note the following: it would be relatively easy to add this capability. However, it should be pointed out that this seems an extremely undesirable requirement. While it is useful as long as a program is running on a specific machine with a known translator, its use can hamper efficient portability. Thus, the programmer might specify an open subroutine for a particular configuration and translator and yet when that program is moved to another machine and/or translator, it might be far more efficient to have a closed subroutine. Since the programmer has already specified which it is to be, the translator is unable to make the determination. It would be far better to leave this decision entirely in the hands of the translator.

K1.COBOL.Operating System Not Required
T (Satisfies)

While most COBOL implementations assume that there is an operating system, there is nothing in the language specifications requiring one. When there is an operating system, the ENVIRONMENT DIVISION and the USE verb provide the appropriate interface between the hardware/operating system combination.

K2.COBOL.Program Assembly
T (Satisfies)

The Inter-Program Communication module of COBOL (Section XII) does allow the creation of separately written modules for both type definitions and executable routines. The COPY statement (pages X-2 through X-4) serves the purpose of incorporating text into a COBOL source program. The COPY statement can be used in any sensible place within a source program (page X-2, point 7). COBOL does not have a language requirement specifying the form of the separately defined program modules.

K3.COBOL.Software Development Tools
Partially Satisfies

COBOL contains a Debug Module which gives the user control over debugging facilities; at compile time, the user is able to specify whether these should be translated for use at object time. Some translators contain the desired tools.

For needed changes, note the following: there are no restrictions in COBOL which would prevent the interface with software tools that might be developed. Support packages could easily be added to the language itself if the requirements were defined.

K4.COBOL.Translator Options
Partially Satisfies

COBOL does not itself contain any options to aid the generation, test, or modification of programs. COBOL does aid documentation by insisting on a particular reference format, and by the inherent readability which is a fundamental aspect of COBOL. Some translators supply these tools but it is impossible to specify exactly what all translators do.

For needed changes, note the following: it should be easy to include additional specifications along these lines once the requirements were determined.

K5.COBOL.Assertions and Other Optional Specifications
Partially Satisfies

COBOL contains a whole Debug Module (Section XI) but does not currently permit inclusions of assertions, assumptions, etc.

For needed changes, note the following: it should not be too difficult to include the missing language elements.

L1.COBOL.No Superset Implementations
Fails to Satisfy

The COBOL standard does permit an implementer to include language features which are not in the ANSI COBOL language "even though...it may prevent proper compilation of some programs that meet the requirements of this standard" (page I-6).

For needed changes, note the following: it would be trivial to change this rule to meet the requirement of L1.

L2.COBOL.No Subset Implementation
Fails to Satisfy

COBOL does not meet this requirement because the standard was specifically designed to permit a modular approach to implementation. Thus, there is a Nucleus and 11 specifically defined language modules (page I-1, section 1.2). Furthermore, many of the modules have subsets and the implementer is essentially able to choose what he wishes.

For needed changes, note the following: obviously it would be trivial to change the current definition of the standard to meet any particular subsetting requirement of TINMAN. There is no problem in the language specification.

L3.COBOL.Low-Cost Translation
Partially Satisfies

There are a great many options available to the COBOL programmer and his use of all of these will obviously increase compile time costs. Conversely, his failure to use many of them will decrease the compile time costs. There are no obvious cases in which the programmer can directly control the trade-off between compile-time and run-time costs. By now there are many efficient compilers of COBOL and hence the language is capable of being compiled fairly efficiently. Nothing can be said about translators for a future language.

For needed changes, note the following: it is not possible to state which changes should be made to allow COBOL to fully satisfy this requirement.

L4.COBOL.Many Object Machines
T (Satisfies)

COBOL 68 has been implemented for almost all full size computers and even one microprocessor. Many features in COBOL 74 were also included the later COBOL 68 implementations. The OBJECT COMPUTER paragraph (pages II-6 through II-7) in the ENVIRONMENT DIVISION makes this specification quite easy for the user.

L5.COBOL.Self-Hosting Not Required
T (Satisfies)

The CONFIGURATION Section in the ENVIRONMENT DIVISION (pages II-5 through II-7) enables the user to specify both the source computer on which the translation will be done and the object computer on which the program is to run. There are COBOL 68 compilers for most computers.

L6.COBOL.Translator Checking Required
Unknown

This is not a language requirement.

L7.COBOL.Diagnostic Messages
Fails to Satisfy

COBOL does not contain a suggested set of error and warning situations for the translator to implement.

For needed changes, note the following: it would be relatively easy to add such suggestions to the language definition.

L8.COBOL.Translator Internal Structure
T (Satisfies)

COBOL does not dictate to a translator the structure or characteristics of a particular translator.

L9.COBOL.Self-Implementable Language
T (Satisfies)

L4.COBOL.Many Object Machines
T (Satisfies)

COBOL 68 has been implemented for almost all full size computers and even one microprocessor. Many features in COBOL 74 were also included the later COBOL 68 implementations. The OBJECT COMPUTER paragraph (pages II-6 through II-7) in the ENVIRONMENT DIVISION makes this specification quite easy for the user.

L5.COBOL.Self-Hosting Not Required
T (Satisfies)

The CONFIGURATION Section in the ENVIRONMENT DIVISION (pages II-5 through II-7) enables the user to specify both the source computer on which the translation will be done and the object computer on which the program is to run. There are COBOL 68 compilers for most computers.

L6.COBOL.Translator Checking Required
Unknown

This is not a language requirement.

L7.COBOL.Diagnostic Messages
Fails to Satisfy

COBOL does not contain a suggested set of error and warning situations for the translator to implement.

For needed changes, note the following: it would be relatively easy to add such suggestions to the language definition.

L8.COBOL.Translator Internal Structure
T (Satisfies)

COBOL does not dictate to a translator the structure or characteristics of a particular translator.

L9.COBOL.Self-Implementable Language
T (Satisfies)

COBOL compilers can be written in COBOL; a very early version of a COBOL compiler was written in COBOL.

M1.COBOL.Existing Language Features Only
T (Satisfies)

Clearly COBOL is currently composed of features within the state-of-the-art. Furthermore, any changes necessary to achieve the requirements of TINMAN are in the category of engineering and not research.

M2.COBOL.Unambiguous Definition
Fails to Satisfy

The semantics of COBOL have not been defined in any formal fashion and currently contain ambiguities as is true of any English language definition.

For needed changes, note the following: probably it would be very difficult, although not impossible, to prepare a formal definition of COBOL semantics.

M3.COBOL.Language Documentation Required
Partially Satisfies

The COBOL standard contains a complete technical description of the language. The syntax is defined formally but the semantics are not defined in a formal fashion. The COBOL standard also contains a fair amount of tutorial or introductory material, even though it is not labelled as such and is scattered through the standard.

The only places in which characteristics of the source language are dependent on the translator, are those characteristics labelled as implementer-defined (pages I-7 through I-8). The current language standard document does not contain examples of each language construct, although it does contain listings of key words and language defined defaults. The standard is well written, and is put together for use by knowledgeable people. It is easy to find specific information in it.

For needed changes, note the following: it would be fairly simple to write tutorials for inclusion as part of the official documentation. There exist books and specific manuals on COBOL which serve this purpose. Instead of adding examples of each language construct, it would be more appropriate to have implementers issue documents on

expensive and inexpensive features. These features tend to vary depending on the translator.

M4.COBOL.Control Agent Required
T (Satisfies)

This requirement refers to the future. However, from its very beginning, COBOL has been "configuration managed". The defining authority has been CODASYL, and its subcommittee has been defined as the group responsible for COBOL. The people developing the standard have long accepted the principle that the only thing that they can do is pick and choose from among the features in the CODASYL-defined version of COBOL. Further, the U.S. Navy and NBS have provided validation services. Thus, there is a lot of experience and precedent in this management concept for COBOL. It seems to have worked fairly well.

M5.COBOL.Support Agent Required
Unknown

This is not truly a language requirement but is a consideration for the future.

A strong standards and configuration management agency required by M4 could presumably perform compiler validation, tools design, library standards and distribution, etc.

M6.COBOL.Library Standards and Support Required
Unknown

This is not truly a language requirement but is a consideration for the future.

A strong standards and configuration management agency required by M4 could presumably perform compiler validation, tools design, library standards and distribution, etc.

Section IVc - COBOL Language Features Not Needed

#.COBOL.Features To Be Eliminated From The Language

Because of its great power in the data handling and input/output area, there are a number of features in COBOL 74 which could be eliminated as unnecessary for TINMAN requirements. Since the COBOL syntax contains many optional phrases within verbs and clauses, it is not practical to list each case which can be eliminated. Thus, the following represents only a summary of the general features which can be eliminated, based on the COBOL 74 ANSI Standard and the TINMAN requirements. This list is not necessarily complete; (i.e., there are other language elements which could be deleted but a truly detailed listing would serve no useful purpose):

IDENTIFICATION Division

Relative I/O Module

Indexed I/O Module

Sort-Merge Module

Report Writer Module

Various editing characters (e.g., credit, debit, zero suppress, check protect)

Figurative constants

Some of the class conditions

Abbreviated combined relation conditions

Specific verbs: ALTER, INSPECT, SEARCH, SET, STRING, UNSTRING

Miscellaneous options in many places.

Section IVd - COBOL Summary and Recommendations

The following table presents the TINMAN requirements by category and the number in each category which COBOL 74 fully, partially, or failed to satisfy, as well as those which are not relevant or applicable. It is not surprising to note that COBOL does best in areas A, B, D and F. They are, respectively, Data and Types; Operation; Variables, Literals and Constants; and Scopes and Libraries. COBOL is poorest in E (Definition Facilities) since COBOL has no definition facilities aside from subroutines and does rather poorly in Category C (Parameters and Expressions). The entire matter of structure of programs and parameter passage is certainly one of COBOL's weakest concepts.

REQUIREMENT		COBOL
A.	Data and Types	
A1.	Typed Language	T
A2.	Data Types	P
A3.	Precision	P
A4.	Fixed-Point Numbers	T
A5.	Character Data	T
A6.	Arrays	T
A7.	Records	T
B.	Operations	
B1.	Assignment and Reference	T
B2.	Equivalence	T
B3.	Relationals	T
B4.	Arithmetic Operations	T
B5.	Truncation and Rounding	T
B6.	Boolean Operations	P
B7.	Scalar Operations - On arrays	T
B8.	Type Conversion - Implicit	F
B9.	Type Conversion - Explicit	P
B10.	I/O Operations	T
B11.	Power Set Operations	F
C.	Expressions and Parameters	
C1.	Side Effects	T
C2.	Operand Structure	T
C3.	Expressions Permitted	F
C4.	Constant Expressions	F
C5.	Consistent Parameter Rules	P
C6.	Type Agreement in Parameters	F
C7.	Formal Parameter Kinds	F
C8.	Formal Parameter Specification	T
C9.	Variable Numbers of Parameters	F
D.	Variables, Literals and Constant	

D1.	Constant Value Identifiers	T
D2.	Numeric Literals	T
D3.	Initial Values of Variables	T
D4.	Numeric Range and Step Size	T
D5.	Variable Types	T
D6.	Pointer Variables	F

E. Definition Facilities

E1.	User Definitions Possible	P
E2.	Consistent Use of Types	F
E3.	No Default Declarations	F
E4.	Can Extend Existing Operators	F
E5.	Type Definitions	F
E6.	Data Defining Mechanisms	F
E7.	No Free Union or Subset Types	T
E8.	Type Initialization	F

F. Scope and Libraries

F1.	Separate Allocation and Access Allowed	F
F2.	Limiting Access Scope	P
F3.	Compile Time Scope Determination	T
F4.	Libraries Available	T
F5.	Library Contents	T
F6.	Libraries and Compools Indistinguishable	T
F7.	Standard Library Definitions	P

G. Control Structures

G1.	Kinds of Control Structures	P
G2.	The GOTO	P
G3.	Conditional Control	P
G4.	Iterative Control	T
G5.	Routines	F
G6.	Parallel Processing	F
G7.	Exception Handling	T
G8.	Synchronization and Real-Time	P

H. Syntax and Comment Conventions

H1.	General Characteristics	P
H2.	No Syntax Extensions	T
H3.	Source Character Set	T
H4.	Identifiers and Literals	P
H5.	Lexical Units and Lines	F
H6.	Key Words	P
H7.	Comment Conventions	P
H8.	Unmatched Parentheses	T
H9.	Uniform Referent Notation	T
H10.	Consistency of Meaning	T

I. Default, Conditional Compilation and Language Restrictions

I1.	No Defaults in Program Logic	T
I2.	Object Representation Specifica-	T

AD-A037 638 IBM FEDERAL SYSTEMS DIV GAITHERSBURG MD F/G 9/2
CANDIDATE LANGUAGE EVALUATION AND RECOMMENDATION REPORT.(U)
1977 F30602-77-C-0009

UNCLASSIFIED NL

2 OF 2
ADA037638

END
DATE
FILMED
4-77

tions Optional

	I3.	Compile Time Variables	T
	I4.	Conditional Compilation	F
	I5.	Simple Base Language	F
	I6.	Translator Restrictions	F
	I7.	Object Machine Restrictions	T
J.	Efficient Object Representations and Machine Dependencies		
	J1.	Efficient Object Code	P
	J2.	Optimizations Do Not Change Program Effect	U
	J3.	Machine Language Insertions	P
	J4.	Object Representation Specification	T
	J5.	Open and Closed Routine Calls	F
K.	Program Environment		
	K1.	Operating System Not Required	T
	K2.	Program Assembly	T
	K3.	Software Development Tools	P
	K4.	Translator Options	P
	K5.	Assertions and Other Optional Specifications	P
L.	Translators		
	L1.	No Superset Implementations	F
	L2.	No Subset Implementations	F
	L3.	Low-Cost Translation	P
	L4.	Many Object Machines	T
	L5.	Self-Hosting Not Required	T
	L6.	Translator Checking Required	U
	L7.	Diagnostic Messages	F
	L8.	Translator Internal Structure	T
	L9.	Self-Implementable Language	T
M.	Language Definition, Standards and Control		
	M1.	Existing Language Features Only	T
	M2.	Unambiguous Definition	F
	M3.	Language Documentation Required	P
	M4.	Control Agent Required	T
	M5.	Support Agent Required	U
	M6.	Library Standards and Support Required	U
Totals		T (Fully Satisfies)	46
		P (Partially Satisfies)	22
		F (Fails to Satisfy)	26
		U (Unknown)	4

Although COBOL is widely used and implemented for its intended class of applications, THERE WOULD PROBABLY BE GREAT PSYCHOLOGICAL RESISTANCE TO ITS ADOPTION AS A BASIS FOR TINMAN WHATEVER COBOL'S TECHNICAL MERITS MIGHT BE WITH RESPECT TO OTHER LANGUAGES.

COBOL 74 considered as a base language for the DoD HOL is:

a. Moderately compliant with existing TINMAN requirements

b. Easily changed to meet approximately 23 of the requirements it does not fully meet

c. Moderately easily changed in approximately 9 of the requirements it does not fully meet

d. Difficult or impossible to change in approximately 13 of the requirements it does not fully meet

e. A viable base technically, but is probably unacceptable psychologically because of its verbosity and past history and the low opinion of it held by language experts.

Note that the numbers above represent complete compliance and include some requirements rated "90%, fully satisfies" in the report.

Section V - HAL/S EVALUATION

Section Va - HAL/S Language Introduction

#.HALS.Introduction

HAL/S is a higher order programming language developed by Intermetrics, Inc. for the flight software of the NASA Space Shuttle program. The language is expressly designed to allow programmers, analysts, and engineers to create software which is reliable, efficient, highly readable, and easily maintained.

HAL/S is intended to satisfy virtually all the flight software requirements of the Space Shuttle. To achieve this, the language incorporates a very wide range of features, including applications oriented data types and computations, real-time control, and constructs for implementing systems programming algorithms.

Data Types and Computations

HAL/S is a linear algebraic language particularly suited for aerospace proramming. Integer, scalar, vector and matrix data types, together with appropriate operators and built-in functions (e.g., a complete vector/matrix library) provide an extremely powerful tool for the implementation of mathematical (navigation, guidance and control) algorithms. Bit and character string processing constructs are available. The formation and use of multidimensional arrays, and of tree-like organizations of heterogeneous data, are also featured.

Real-Time Control

HAL/S is a real-time control language. With time being a critical parameter in many on-board solutions, time dependence has been introduced into HAL/S as part of the syntax. A wide range of commands for controlling real-time tasks is provided including one-shot and cyclic scheduling based on time, priority and hardware and/or software events.

Program Reliability

A major goal of the HAL/S design is the production of reliable software. Effective isolation between separately compiled program blocks contributes to modularity while communication is supervised

through centrally managed and highly visible common subroutines and data pools. Access to common resources may centrally be granted or restricted. For a real-time environment, HAL/S includes a locking machanism which can automatically protect shared data and code.

Error Recovery

HAL/S has a comprehensive and flexible mechanism for detecting and recovering for run-time errors. It also has the capability of simulating run-time errors, which can be extremely useful for checkout purposes. Another feature of the language is the ability to specify and signal user-defined run-time errors.

Every active real-time process possesses its own so-called "error environment", which is essentially a description of the recovery actions in force for all possible run-time errors the process could be subject to. On initiation of the process, the system recovery action is in force for all run-time errors. During the life of a process, its error environment may be modified by the specification of a "user recovery action" for some error or error group. The user recovery action is enforced by the execution of specific HAL/S error control statements.

Systems Programming Features

HAL/S contains a number of features especially designed to facilitate its application to systems programming, thus substantially eliminating the necessity of using an assembler language. Two of these features are the facility to create and manipulate data pointers and to create language extensions in the form of "%macros".

a. Data Pointers - In HAL/S, pointer data items are called "NAME" data items. To substantially eliminate software unreliability, a specific mechanism in HAL/S assures that any given NAME data item can only point to other data items of a single kind specified at compile time. The mechanism consists in declaring a NAME data item with properties of type, precision, and arrayness, just as if it were an ordinary data item. These properties, rather than actually belonging to the NAME data item, are the properties which must be processed by data items to which the NAME data item can point.

b. %Macros - Realizing that it may be necessary to provide capabilities not basic to the language, HAL/S isolates these "system extensions" to a set of compiler-supplied %macros. These are written in assembler language and may be designed to meet specific requirements and interfaces. %Macros are

incorporated in the compiler run time library or inserted as in-line code. As with other compiler "built ins", they benefit from the automatic checking conducted at compile time to assure proper usage. In effect, the %macro provides a means of adding functional, special-purpose implementation-dependent extensions to HAI/S without requiring far reaching syntax changes or substantial modification of the compiler programs.

Below are listed some of the strong points of HAL/S:

1. Real-time features include multitasking, event processing, and shared data management.

2. Built-in data types and operators for vector and matrix data are useful in programming spacecraft guidance and control systems.

3. Structured output listings with indentation for nesting levels are very readable and are somewhat self-documenting.

4. It is a block-oriented language with standard name scoping features. Separately compiled program blocks can be executed together and can communicate through one or more highly visible data pools (compools), which are themselves separately compilable.

5. Reliability is enhanced through strong type checking and structured programming constructs.

6. An error recovery facility allows the programmer freedom to define his own error processing logic or leave control with the system.

Some of the weak points of HAL/S:

1. There is a lack of "open-ended" capabilities - no extensibility, no generic procedures, and no recursion. It shall be noted that these are intentional omissions because of time and space restrictions inherent in an embedded, man-critical system.

2. There is a basic lack of versatility in the I/O capabilities.

3. No specific symbol is used for multiplication; the adjacency (with a space) of two operands indicates multiplication.

Following is an itemized comparison of HAL/S against the DoD TINMAN requirements for a High-Order Language (HOL). An additional paragraph gives a brief discussion of the source of modifications required to fulfill TINMAN requirements. The reference document used is the "HAL/S Language Specification", Intermetrics Inc., Version IR-61-8, June 16, 1976. Each major section is prefaced with general comments and a summary.

A.HALS.Data and Types

HAL/S is a strongly typed language with a diverse set of data types. All data in a HAL/S program is defined by DECLARE statements. The group of all DECLARE statements valid for a program scope appear at the beginning of that scope in the data declaration block.

HAL/S adequately fulfills the TINMAN requirements for a strongly typed language, with two exceptions. There is no fixed-point capability and character sets are not treated as enumeration types. Additional capabilities include the EVENT, BIT, NAME, MATRIX, and VECTOR data types - each with their own set of characteristics and operation.

A1.HALS.Typed Language
T (Satisfies)

HAL/S requires all data to be declared and fully qualified in the source program (Section 4.0, page 4-1 through Section 4.7, page 4-22). There exists a set of default properties for all data items (e.g., floating-point, single precision, non-initialized) which HAL/S assumes if none is given (see Lang. Spec. page 4-19). But the data identifier must appear in a declaration statement for this to occur. HAL/S does not allow dynamic declaration.

A2.HALS.Data Types
Partially Satisfies

HAL/S provides integer (INTEGER), real-floating-point (SCALAR), Boolean (BOOLEAN), and character (CHARACTER) basic data types. Five additional types are also provided: BIT, VECTOR, MATRIX, NAME, and EVENT (Section 4.7, page 4-19). The BIT designation is simply a string of Booleans. The VECTOR and MATRIX designation defines a specified (or, if unspecified, a default) number of floating-point values which are naturally related. The NAME is a pointer variable which can be manipulated. The EVENT designation is a specially defined Boolean with

real-time programming implications. All HAL/S data items listed above may be defined in groups of homogeneous types called ARRAYs (Section 4.5). All data items or arrays of data items may be defined in groups (copies) of generally non-homogeneous types called STRUCTURE's (Section 4.3). Indexing to the STRUCTURE copy, ARRAY element and, in the case of CHARACTER, BIT, MATRIX, and VECTOR, to the component level, is provided.

Modification required: modification of HAL/S to provide fixed-point capability is moderately difficult. HAL/S has available a simple fixed-point capability which is currently being used on some applications. This facility is designed for machines which have no floating-point hardware. Although currently there is no implementation of HAL/S which has both fixed-point and floating-point capabilities, it is feasible to have such. The compiler modifications (library routines, additional type checks, etc.) to provide such a combination of capabilities would be moderate. This evaluation has deemphasized the presence of this additional feature in analysis of individual requirements dealing with fixed-point capabilties.

A3.HALS.Precision
T (Satisfies)

HAL/S does not provide the capability to specify or change precision globally (on the scope level). It does permit precision specification for individual variables at DECLARE time (Section 4.7, page 4-20, general rules 2 and 3). All floating-point and integer arithmetic is done in single precision unless one or more of the operands have been explicitly declared to be double precision or unless an explicit conversion is requested in the expression (Section 6.1.1, page 6-4).

A4.HALS.Fixed-Point Numbers
Fails to Satisfy

HAL/S, as specified in the reference documents, has no fixed-point capability.

Modification required: modification of HAL/S to provide fixed-point capability is moderately difficult. HAL/S has available a simple fixed-point capability which is currently being used on some applications. This capability is designed for machines which have no floating-point hardware. Although currently there is no implementation of HAL/S which has both fixed- and floating-point capabilities, it is feasible to have such. The compiler modifications (library routines, additional type checks, etc.) to provide such a combination of capabilities would be moderate. This evaluation has deemphasized the

presence of this additional feature in analysis of individual requirements dealing with fixed-point capabilties.

A5. HALS.Character Data
Fails to Satisfy

HAL/S does not consider character strings as enumeration types, nor does it provide the capability to specify an "order of characters" at compile time. A single character set is defined for the language. This character set is in itself an enumeration type, with an implied collating sequence and subject to all relational operators (see B3).

Modifications required: to add the capability for sets of enumerated types and to include character sets would be a difficult modification.

A6.HALS.Arrays
T (Satisfies)

HAL/S provides for up to n dimensions (where n is implementation-dependent) in ARRAY declarations (Section 4.5, page 4-14, rule 2). The dimension number(s) specify the upper bound of the range of each dimension. (The lower bound is understood to be 1.) The dimension value may be expressed as an arithmetic expression, but is rounded to the nearest integer value. The uppermost bound of an array which appears as a parameter of a Procedure or Function may be determined at entry into the Procedure/Function, for one dimensional arrays only (Section 4.5, page 4-16, restriction #1).

A7. HALS.Records
T (Satisfies)

The STRUCTURE facility in HAL/S fulfills this requirement (Section 4.3). It provides for a hierarchical organization of generally non-homogeneous data types. The NAME facility (described under TINMAN D6) applied to structures, provides an "overlay" capability with type checking.

B. HALS.Operations

HAL/S has three basic categories of operators (corresponding to three categories of data types): arithmetic, Boolean, and character. The largest number of operators fall into the arithmetic category. HAL/S uses a standard symbol with infix notation for all operations.

HAL/S adequately fulfills all requirements to provide operations to be used by the data types described in section A. The single exception is the requirement for operations on "power sets of enumeration types". Other weaknesses in the operation category include implicit rounding and truncation in some cases, implicit type conversion in some cases, and no dynamic I/O device assignment. HAL/S exceeds TINMAN operation requirements in that it provides built-in operations for vectors and matrices.

B1.HALS.Assignment and Reference
T (Satisfies)

HAL/S provides automatic assignment operations as a part of the definition for all data types. Any variable data item may have the value of another variable, a constant value, or the result of an expression assigned into it, providing the data types are equivalent. There are, however, a few exceptions to data type equivalency (Section 7.3, page 7-6, general rule 5). The uniform symbol for assignment for all data types is the "=" sign (Section 7.3). Composite data types (e.g., arrays, structures, matrices) may be treated as a single entity in assignment or reference (Section 7.3, page 7-5):

```
ARRAY1 = ARRAY2 + ARRAY3;
```

Additionally, HAL/S has a multiple-assignment capability which allows the value of a single operand to be stored into several variables in one statement (Section 7.3, page 7-5):

```
FIRST, I, LOOP-COUNTER = 1;
```

B2.HALS.Equivalence
T (Satisfies)

The symbol for identity or logical equivalence is the "=" sign. (See comments in H10.) Equivalence comparisons must be between like data types. Arithmetic expressions cannot be implicitly compared to Booleans; nor character strings to arrays, etc. However, mixed expressions of integer and floating-point operands are accepted for arithmetic equivalence comparison (Section 6.2.1). Mixed precision is also accepted within arithmetic comparisons. HAL/S also has equivalence comparison capability between aggregate data items such as arrays, structures, vectors, and matrices. An array can be compared to a single item (of the same type) for identity. (Identity exists only if all elements of the arrayed operand are equal to the unarrayed operand.) An array may also be compared to another array of identical dimension for equivalnce (Section 6.2.5). Evaluation of equivalence for each

corresponding pair of elements of the operands holds true in comparing vectors, matrices, character strings, Boolean strings and structures (Section 6.2.1, rule 4; Section 6.2.2, rule 3; Section 6.2.3, rules 1 and 4; Section 6.2.4, rule 4).

B3.HALS.Relationals
T (Satisfies)

All arithmetic data types in HAL/S have defined for them six basic relational operators ("=", " =", ">", ">=", "<", "<="). The [algebraic not sign] is the symbol for NOT. Alternatively the keyword "NOT" may be used in conjunction with "=", "<", ">". The relational operators "=" and "NOT=" may also be applied to the VECTOR, MATRIX, BOOLEAN, ARRAY, and STRUCTURE data (Section 6.2.1).

B4.HALS.Arithmetic Operations
Partially Satisfies

HAL/S has the set of arithmetic operations listed, although it uses a space or blank instead of a symbol for multiplication (Section 6.1.1, page 6-3). Integer division with remainder is not provided in the set of infix operators. The user can obtain the results of integer division by way of two built-in functions. The function DIV (alpha, beta) (where alpha and beta are integers) yields an integer result with no remainder (Appendix C, page C-1). The function REMAINDER (alpha, beta) (where alpha and beta are integers) yields the integer remainder resulting from the division of beta into alpha (Appendix C, page C-1).

Standard multiplication is provided for all data types, including vectors and matrices. Additionally, HAL/S provides a dot product and a cross product operation for vectors (Section 6.1.1, page 6-4). Operations involving vectors or matrices compile into machine code which does the specified operation iteratively on each element. However, this provides the programmer a cleaner listing and often times more concise machine code.

Modifications required: the capability to provide integer division with remainder as an infix operator could be provided with no difficulty.

B5.HALS.Truncation and Rounding
Partially Satisfies

A certain degree of implicit rounding or truncation takes place in

assignment operations between arithmetic values of different types or precision (Section 7.3, page 7-6, semantic rules):

1. Assignment of an integer expression into a floating-point variable results in an implicit conversion with a possible loss of decimal places of accuracy.

2. Assignment of a floating expression into an integer variable results in rounding to the nearest integral value. This may cause an execution exception if the absolute value is too large to be represented as an integer.

3. Assignment of a double precision floating-point value into a single precision floating-point variable will result in truncation of binary digits from the mantissa.

4. Assignment of a double precision integer value into a single precision integer value will result in truncation of the high-order digits. (Only the low order bits are stored.)

Modifications required: the default rounding, truncation and implicit conversion could be eliminated, but not without some difficulty (moderate). Explicit calls to conversion routines could easily be replaced with calls to diagnostic routines. However, many implicit conversions peculiar to the object machine would have to be anticipated and prefaced with calls to diagnostic routines.

B6.HALS.Boolean Operations
T (Satisfies)

HAL/S Boolean operators include "AND", "OR", and "CAT" (Section 6.1.2, page 6-7). The CAT operator is used to concatenate strings of Booleans. The logical complement of a Boolean expression may be obtained by prefacing it with "NOT" (Section 6.1.2, page 6-9). There is no infix operator for exclusive OR but a built-in function XOR (alpha, beta) (where alpha and beta are bit strings) provides this capability (Appendix C, page C-8). The "short circuit" mode exists for all connective Boolean operations between relational expressions. (Reference 6.1.2)

B7.HALS.Scalar Operations
T (Satisfies)

HAL/S permits assignment and operations on arrays (Section 6.1.5;

Section 7.3, page 7-5, rule 3) and structures (Section 6.1.4; Section 7.3, page 7-8) (records) of identical format. The operations only appear to provide a parallel operation on all elements simultaneously, while the machine code produced may perform an element-by-element operation.

B8.HALS.Type Conversion
Fails to Satisfy

HAL/S allows many implicit type conversions (Section 7.3, page 7-6 and 7-7; Appendix D).

Explicit conversion between all data types is provided for in HAL/S. For conversion to bit strings from character strings and vice versa, the introduction of a radix is allowed (Section 6.5.2, page 6-32; 6.5.3, page 6-34). This radix causes the conversion to be done in a specified number base:

CHARACTER[@DEC] (BIN'101101') = '45'   where [@] indicates a subscript

An additional capability is the SUBBIT pseudo-variable (Section 6.5.4) which allows access to bit representations within other data types:

SUBBIT[33 to 64] (DP)   where DP is a double precision floating-point value, yields bits 33 through 64 of the machine representation of DP and makes it look like a 32-bit variable.

HAL/S also provides explicit precision conversion between floating-point and integer values (Section 6.6).

Modifications required: the default rounding, truncation and implicit conversion could be eliminated, but not without some difficulty. Explicit calls to conversion routines could be replaced with calls to diagnostic routines. However, many implicit conversions peculiar to the object machine would have to be anticipated and prefaced with calls to diagnostic routines.

B9.HALS.Changes in Numeric Representation
Partially Satisfies

All constant values are checked at compile time for a machine dependent maximum. HAL/S has no compile time warning for possible range errors within expressions, but does have at run time. Explicit conversion of a double-precision integer to single precision will not result in a run time exception. See B5 for other comments on possible error conditions connected with implicit type and precision conversion. (Reference Appendix D)

Modifications required: the addition of a fixed-point capability (see comments under A2) which uses range specification at declare time could certainly include compiler checks for range errors.

B10.HALS.Input/Output Operation
Partially Satisfies

HAL/S provides capabilities for sequential and random access I/O (Section 10.1; 10.2). The sequential I/O commands may be accompanied by control commands which cause explicit movement or positioning of device mechanism (Section 10.1.3). For specific I/O device interface the "%macro" capability may be used. (See comments on J3.) There is no operation which allows the program to dynamically assign or reassign I/O devices.

Modifications required: to provide all of the I/O capabilitis required, as a part of the standard language, would be difficult.

B11.HALS.Power Set Operations
Fails to Satisfy

HAL/S has no data types defined as power sets of enumeration types.

Modifications required: HAL/S would need to provide a subrange capability for enumeration types, allow these sets to be mapped to bit strings and provide special operations to manipulate them. The operations are in place for standard bit strings, but the capability would have to be expanded to encompass all enumeration types. This language change would be extremely difficult.

C.HALS.Expressions and Parameters

HAL/S has a set of rules that governs combining operands and operators to form expressions. The user (or reader) can easily determine the meaning of any expression from the written statement.

Also, there are strict rules for dealing with formal parameters of procedures and functions (e.g., agreement of type and number).

The requirements for expressions and parameters are basically fulfilled by HAL/S. There is no "generic procedure" capability, but this is a debatable requirement. If the intent of this requirement is to handle arrayed arguments or lists of output data, HAL/S adequately provides for these through built-in functions and standard I/O statements.

C1.HALS.Side Effects
T (Satisfies)

HAL/S has a definite precedence and evaluation order for operations in a single expression (Section 6.1.1, page 6-5). The rule of left-to-right evaluation is overridden for the multiplication operators when necessary to summarize the total number of elemental multiplications required (page 6-5, rule 3). However, the user may explicitly alter the hierarchy of evaluation order through use of parentheses.

C2.HALS.Operand Structure
T (Satisfies)

Operators and operands are clearly distinguishable in HAL/S expressions, except for the arithmetic multiply operator which is denoted by a space. (See comments on H10.)

HAL/S has six levels of operator hierarchy. The insertion of two vector operators causes what might be considered a large table of precedence. The levels of precedence are (Reference 6.1.1):

1.	exponentiation	**
2.	multiplication	blank
3.	cross-product	*
4.	dot-product	.
5.	division	/
6.	addition	+
7.	subtraction, negation	-

C3.HALS.Expressions Permitted
T (Satisfies)

HAL/S allows expressions of any resultant type to appear in any

context in which a variable or constant of that type might appear (Section 6.1, page 6-6; Section 5.3.2, page 5-11). In particular, arithmetic expressions or normal functions - either "built-in" or user-defined - may be used as subscripts (see diagrams on page 5-11 and 6-6; Section 6.4, pages 6-23).

C4.HALS.Constant Expressions
T (Satisfies)

HAL/S evaluates all constant expressions at compile time (Appendix F).

C5.HALS.Consistent Parameter Rules
T (Satisfies)

All parameters of a specific type are treated consistently.

C6.HALS.Type Agreement in Parameters
T (Satisfies)

HAL/S does require arrayed formal parameters to have a specified number of dimensions. However, only single-dimension arrays may have the size of their dimension determined at entry into the array scope (Section 4.5, page 4-16).

C7.HALS.Formal Parameters
T (Satisfies)

HAL/S adequately fulfills the intent of this requirement, although there is no "class distinction" made between formal parameters. A formal class of parameters for specifying control action for exceptions does not exist. (See comments on G7.) Actual procedure parameters, although, not a separate "class", must be declared initially in the procedure and must agree in type and attributes with those of the calling arguments (Section 3.7.2, 3.7.3).

C8.HALS.Formal Parameter Specification
Fails to Satisfy

HAL/S requires the re-declaration of all formal parameters with their associated attributes in the procedure or function body. Generic procedures are not possible in HAL/S. Compile time type checks are performed on all formal parameters (Section 3.7.2).

Modifications required: see comments under C9.

C9.HALS.Variable Number of Parameters
Fails to Satisfy

HAL/S will not allow a variable number of input arguments to procedures or functions (Section 3.7.2).

Modifications required: certainly the concept of allowing a variable number of arguments on procedure calls and optional format parameter definition within the procedure itself is feasible. However, the compiler modifications necessary to implement this capability would be extremely difficult to make.

D.HALS.Variables, Literals, and Constants

HAL/S accomplishes the need to associate initial values with all data types. With the single exception of fixed-point range and step size specification, HAL/S meets or surpasses all requirements in this area.

For the modifications required to fully comply with TINMAN, see comments under section A for fixed-point discussion.

D1.HALS.Constant Value Identifiers
T (Satisfies)

HAL/S provides the capability to associate constant values with identifiers and has compiler checks preventing changes to them. This is specified with the CONSTANT attribute (Section 4.8).

D2.HALS.Numeric Literals
Unknown

HAL/S' compliance with this requirement is unknown from defined references.

D3.HALS.Initial Value of Variables
T (Satisfies)

The INITIAL capability may be used with any declared identifier

(Section 4.8), but if not used, generally no default value is assigned to it. The exceptions to this are the NAME pointer which defaults to <null> (Section 11.4.10, page 11-34), and the EVENT variable which defaults to FALSE (Section 4.8, page 4-26).

The INITIAL capability may be used in connection with the keywords AUTOMATIC or STATIC (Section 4.5, page 4-14). The AUTOMATIC attribute causes an identifier with the initialization attribute to be initialized on every entry into the code block containing its declaration. The STATIC attribute causes an identifier to be initialized only on the first entry into the code block. Thereafter its value is guaranteed to be preserved for the next entry into the block. If neither attribute is specified, then STATIC is assumed.

D4.HALS.Numeric Range and Step Size
Fails to Satisfy

HAL/S does not have the capability to specify range and step size in its limited fixed-point feature. (See comments under A4.)

D5.HALS.Variable Types
Partially Satisfies

Any built-in data type may be associated with a variable or array (Section 4.5, page 4-13; Section 4.7, page 4-19). Components of structures may be structures or arrays (Section 4.3, page 4-10; Section 4.5). Structures may be multicopied, which gives them an array-like appearance (page 4-22). HAL/S has no user-defined types or subsequences of enumeration types to which this requirement could be applied (see comments in section E).

D6.HALS.Pointer Variables
T (Satisfies)

The NAME facility is a declared attribute which provides the pointer capability. To greatly eliminate software unreliability sometimes inherent in a pointer mechanism, HAL/S requires that NAME data items can only point to other non-pointer data items of a specific type defined at compile time. This is accomplished by declaring a NAME data item with properties of type, precision and "arrayness" just as if it were an ordinary data item. These properties, rather than actually belonging to the NAME data item, are the properties which must be possessed by data items to which the NAME data item can point. Type checking is done on all NAME data item manipulations (Section 11.4, page 11-16).

E.HALS.Definition Facilities

HAL/S does not have an extensibility feature as such. In the sense that user-defined functions and structures comprise "extension", HAL/S does comply with the intent of this requirement. If one purpose of requiring extended data types and operations is to provide a capability for creating and manipulating vectors and matrices, HAL/S has these features built-in and has no need for extension in this area.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult. A need for true extensibility has not been seen so far in any usage of HAL/S. The nature of the compiler-builder and ease of modification of the compilers has provided all the flexibility needed.

E1.HALS.User Definitions Possible
Partially Satisfies

HAL/S has a Function capability which defines a "new" operation of the user's own creation. However, it is not directly associated with a particular data definition, nor can it be used as an infix operator. The function invocation may appear within an expression (see comments on H9) giving the same results that would have been obtained if a user-defined infix operator had been specified.

The capability to add machine-dependent extensions to the language is provided in the "%macro" (Section 11.2.2). The invocation of a %macro may cause the compiler to generate inline object code to be executed directly or it may generate linkages to external routines. In neither case does the %macro act as a new user-defined data type or infix operator. However, it does provide some of the flexibility demanded by language extension.

The user data definition facility in HAL/S is even less obvious. The structure definition (Section 4.3) and manipulation capabilities partially fulfill the intent of user-defined data (see A2, A7, B1, B2, and B3).

HAL/S also has a REPLACE capability which can be used to define an identifier text substitution which is to take place wherever the identifier is referenced (Section 4.2, page 4-2 through 4-7.1).

An example would be:

```
REPLACE INCR(X) BY "X=X+1";
```

Thus, a simple increment function has been defined for the name scope of the definitions.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E2.HALS.Consistent Use of Types
Fails to Satisfy

Even assuming the "extension features" described under E1, HAL/S does not fulfill this requirement.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E3.HALS.No Fault Declarations
Fails to Satisfy

HAL/S allows many default attributes in data declarations (e.g., 3-vector, 3x3 matrix, floating-point single precision, etc.) (Section 4.7, pages 4-19 and 4-20).

Modifications required: the default attributes provided by the compiler could be easily removed and a compile-time error condition flagged for all data not fully declared and qualified.

E4.HALS.Can Extend Existing Operators
Fails to Satisfy

HAL/S has no capability to extend the meaning of operators.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E5.HALS.Type Definition
Fails to Satisfy

HAL/S has no capability for user data type definition.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E6.HALS.Data Defining Mechanisms
Fails to Satisfy

HAL/S has no capability for user data type definitions.

To make the HAL/S languge extensible by allowing user definitions of data and operations would be very difficult.

E7.HALS.No Free Union or Subset Types
T (Satisfies)

This is a negative requirement. HAL/S fulfills this requirement by default since there are no user data type definitions.

E8.HALS.Type Initialization
Fails to Satisfy

HAL/S has no capability for user data type definitions.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

F.HALS.Scopes and Libraries

HAL/S completely fulfills the requirements for language scopes and libraries and modifications are unnecessary.

F1.HALS.Separate Allocation and Access Allowed
T (Satisfies)

The ACCESS attribute may be applied to declared data, causing implementation-dependent managerial restrictions on access to be placed on the variable when it is used in assignment contexts (Section 4-5, page 4-15). ACCESS may also be applied to a program block (procedure or function) (Section 3.7.1, page 3-14.1).

The UPDATE block is used to control the sharing of data by two or

more real-time processes (Section 3.4, page 3-8). The variables being controlled must have in their declaration the attribute LOCK (n) - where "n" indicated a "lock group" (Section 8.10). The update block is an explicitly delimited body of code whereby the interraction of code blocks referencing or modifying locked data may be controlled.

F2.HALS.Limiting Access Scope
T (Satisfies)

The ACCESS capability as applied to data, compools, programs, procedures, or functions fulfills this requirement. (See comments under F1.)

F3.HALS.Compile Time Scope Determination
T (Satisfies)

HAL/S follows a rigid set of rules for determining the scope of identifiers for reference (Section 3.8). Identifiers may be referenced only in their "name-scope" (the code block containing the declaration, and all code blocks nested therein).

F4.HALS.Libraries Available
T (Satisfies)

HAL/S provides the capability to build a program complex from several units of compilation (program, external procedure, external function, or compool) (page 3-1). These units may be compiled independently in sequential invocations of a compiler or they may be drawn from implementation-dependent libraries. Any compilation unit referencing another unit must be preceded by a block template describing the referenced unit (Section 3.1, page 3-3; and Section 3.6). These block templates may be contained in implementation libraries and included via compiler directive.

F5.HALS.Library Contents
T (Satisfies)

HAL/S' programs may call routines written in other languages. The invocation of this capability is noted by the NONHAL keyword followed by an integer number indicating the type of program unit to be accessed (Section 4.6). The NONHAL attribute attached to a function or procedure name indicates to the compiler that a predefined linkage mechanism should be set up for that unit. Current implementations use the NONHAL capability to interface with 360 Assembler and FORTRAN code blocks.

However, the capability is open ended and could easily be expanded to other languages.

F6.HALS.Libraries and Compools Indistinguishable
T (Satisfies)

HAL/S does not preclude the generation of libraries which may include programs, procedures, functions or compools. The compool may contain data of all types and may be associated with any unit of compilation in order to share data (Section 3.5). The control and access to completion units within libraries is accomplished with the "block template" facility (Section 3.6).

F7.HALS.Standard Library Definitions
T (Satisfies)

HAL/S provides standard capabilities for sequential and random access I/O (Section 10.1 and 10.2). Several miscellaneous machine-independent built-in functions provide machine-dependent capabilities (e.g., CLOCKTIME - time of day, RUNTIME - real-time executive clock time, DATE, etc.) (Appendix C, page C-5).

G.HALS.Control Structures

HAL/S provides all of the basic control structures required for controlling and altering program flow. It does not provide true recursion. The Boolean expression conditional control structure is not required to be fully partitioned, nor are control variables required to be local to the loops.

G1.HALS.Kinds of Control Structures
Partially Satisfies

HAL/S does not have control mechanisms for recursive control. There is a set of primitive control structures which provide all the other desired control mechanisms (Section 7.2, 7.6.2 through 7.6.6, 7.7).

See comments under G5 for "modifications required".

G2.HALS.The GOTO
T (Satisfies)

HAL/S has a GOTO feature which causes a branch in execution to a labeled executable statement within the same name scope. The GOTO may not cause execution to branch into a DO...END group, or into or out of a code block (Section 7.7).

G3.HALS.Conditional Control
Partially Satisfies

HAL/S provides all three conditional control structures mentioned in this requirement: IF...THEN...ELSE (Section 7.2), discrete DO FOR (Section 7.6.4), and the DO CASE (Section 7.6.2).

HAL/S does not require the IF...THEN...ELSE or the DO CASE to be fully partitioned. If the ELSE clause is presented in an IF...THEN...ELSE construct, it may not be preceded by an embedded IF...THEN statement without an ELSE clause:

```
IF (condition)

THEN

          IF (condition)       embedded IF...THEN
          THEN                 not allowed.

ELSE (statement)
```

This prohibits the occurrence of the "dangling ELSE" problem. The embedded IF...THEN can be used if enclosed in a DO...END group.

. Modifications required: to force an ELSE clause to be present in every IF...THEN...ELSE and DO CASE construct would be a relatively simple compiler modification. These clauses are implied by the very nature of the control structures and are conceptually present, though not seen. Either a compiler diagnostic message for omission or insertion of an explicit "ELSE <null>;" would be possible to satisfy this requirement.

G4.HALS.Iterative Control
Partially Satisfies

Control variables are not required to be local to the iterative control structure. An optional capability to provide temporary integer

variables (loop control counters, etc.) within the control structure is present in the TEMPORARY data declarative attribute (Section 11.3).

HAL/S provides the capability to specify more than one terminating predicate:

DO FOR (iterative control) WHILE (condition); (Section 7.6.2)

or

DO FOR (iterative control) UNTIL (condition); (Section 7.6.3)

Modifications required: actually no compiler modification is required for this capability. The TEMPORARY feature fulfills it completely. However, the programmer has an option to use this feature or to use non-local variables for loop control. A simple compiler modification would be to interpret all "DO FOR" constructs to be "DO FOR TEMPORARY".

G5.HALS.Recursive Routines
Fails to Satisfy

True recursion (invoking a routine from within itself) or co-routine recursion (calling routine B from routine A, and B, in turn, calling A again), is not allowed in HAL/S (Programmers' Guide, page 1-5).

Modifications required: the mechanism to make HAL/S recursive is basically present in the form of the totally reentrant capability. Addition of run-time checks for extreme stack sizes (caused by possible infinite loops) could be made.

G6.HALS.Parallel Processing
T (Satisfies)

HAL/S has a facility for creating a multiprocessing job structure in a real-time programming environment. The Real-Time Executive (RTE) controls the execution of processes held in a process queue. Processes may be in one of four states in the process queue - active, wait, ready, stall. Depending on the implementation, it is possible for several processes to be active (in execution) simultaneously (Section 8.1, page 8-2; Programmers' Guide, pages 3-2 and 13-4). HAL/S contains statements which schedule processes (enter them into the process queue - Section

8.3), terminates them (removes them from the queue - Section 8.4 and 8.5), and otherwise directs the RTE in its controlling function. Several processes may be put into the active state simultanously by using the SCHEDULE statement and specifying immediate activation (Diagram, page 8-4; items 7 and 8, page 8-6).

G7.HALS.Exception Handling
T (Satisfies)

HAL/S allows the user to change the prevailing error recovery action via the ON ERROR command (Section 9.1). When an error occurs, the action to be taken, either system- or user-defined, is determined as a result of the ON ERROR statement in force at the time. HAL/S has no provision for passing data parameters to a recovery point. However, the error recovery system has access to information such as statement number, run time and trace back through calling sequence, that might be considered "parameterized data" (Section 9.0, second paragraph).

G8.HALS.Synchronization and Real-Time
T (Satisfies)

The WAIT capability (Section 8.6) fulfills this requirement by causing delay in execution for a time delta or until a certain notice event has been set (process event or I/O). HAL/S allows programs to execute in pseudo-parallel via the SCHEDULE statement (Secton 8.3, diagram). This statement has provisions to delay beginning execution until an event has taken place, at a certain time, or after a specified time delay. Priority of execution and repetition rates may be specified. Termination criteria (e.g., after an event, or time) may also be stated in the SCHEDULE statement (Section 8.3).

H.HALS.Syntax and Comment Conventions

Basically HAL/S has a very powerful and versatile syntax. The input is "stream-oriented", that is, statements may begin anywhere on a line (or card) and may overflow onto succeeding lines or cards. Certain input columns are reserved for special control information (see H5).

H1. HAL.General Characteristics
T (Satisfies)

HAL/S is a free-format language with statements made up of literals (Section 2.3.3), user-defined variables (Section 2.3.2), language unique keywords (Section 2.3.1), and certain special characters (see table on page 2-4). Blanks may be interspersed between most of the elements

making up a source line (Section 2.5). HAL/S allows up to 32 characters in identifiers with break characters interspersed as desired to enhance readability. The specific statement delimiter used is the semicolon. No keyword abbreviations are allowed.

H2.HALS.No Syntax Extensions
T (Satisfies)

HAL/S does not allow the user to modify the syntax of the source language.

H3.HALS.Source Character Set
T (Satisfies)

Following is a list of the character set used in HAL/S. It is a subset of the standard ASCII and EBCDIC sets.

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
+-*./ &=<># $,;:'")(%
(blank)

The only HAL/S feature inaccessible using the 64 character ASCII subset only is the "escape mechanism". This capability allows the user to introduce within a character literal, a character not included in the HAL/S character set (Section 2.3.3, page 2-10, rule 5). It is introduced with the "[cent]" which is not a part of the basic 64-character ASCII subset.

The character set is easily modified by substitution of acceptable characters on an implementation basis. This would allow use of an acceptable ASCII character for the "[cent]" sign.

H4.HALS.Identifiers and Literals
T (Satisfies)

Very precise syntactical rules for identifiers (Section 2.3.2) and literals (Section 2.3.3) are specified for HAL/S. The break character used is the underscore. The space or blank is usually a delimiter unless it is embedded in a character string or is meant to signify multiplication (see B4 and H10). The insertion of a break character within an identifier name causes a unique identifier to be created (e.g., ALPHA-BETA is not identical to ALPHABETA).

H5.HALS.Lexical Units and Lines
Partially Satisfies

HAL/S partially violates this requirement by allowing the continuation of character literals across input lines. All other lexical units must be contained on a single input line.

HAL/S allows a mixed single or multiline input format. The single line format allows the user to denote exponentiation (**) and subscripting ($) on the same line (Section 2.4). Thus, the expression

$$A = X^{(Y+2)} + B_{(Z-2)}$$

would be written A = X** (Y+2) + B$ (Z-2); in single line format. The multiline format allows the user to input the subscripts and exponents as they appear mathematically -- below and above the main line respectively. The first character position of each line is reserved for a symbol (E,M,S) denoting the exponent, main, or subscript lines. The previous example would be written for multiline input:

```
E        Y + 2
M   A = X      + B    ;
S                Z - 2
```

Modifications required: the capability to allow embedded comments and character strings to continue across input lines could be easily modified to allow concatenation of literal strings implicitly and thus eliminate the one case in which HAL/S violates this requirement.

H6.HALS.Key Words
T (Satisfies)

HAL/S has both keywords (Appendix B) and built-in function names (Appendix C) which are not available for any use other than that which is intended by their definition in the language specification. There are 95 keywords and 56 built-in function names for a total of 151 reserved words.

H7.HALS.Comment Conventions
Partially Satisfies

HAL/S allows embedded comments within a source statement (Section 2.5). The format is:

/* ... Any Text ... */

Such comments may appear between statements, but may not appear in a literal, reserved word, or identifier. The comment is terminated by a delimiter rather than the end of a source line. There are implementation-dependent restrictions on the overflow of embedded comments from line to line of the source text.

Additionally, HAL/S allows a single line comment which has no executable statements in it. A "C" in column one of the input line indicates the beginning of the comment; it is terminated at end-of-line (Programmers' Guide, page 2-11).

H8.HALS.Unmatched Parentheses
T (Satisfies)

No unmatched parentheses, brackets, or braces are allowed in HAL/S; nor are unbalanced DO...END pairs allowed.

H9.HALS.Uniform Notation
T (Satisfies)

HAL/S allows function invocation to occur anywhere that the resultant quantity returned from the function would be valid (Section 6.4).

H10.HALS.Consistency of Meaning
Fails to Satisfy

Depending on the context in which it appears, HAL/S uses the "=" sign to indicate either assignment or equality (Section 6.2.1, 7.3).

The blank character may also have different meanings, depending on the context in which it is used. It may be used as a symbol separator or a delimiter but may also be used to signify multiplication (Section 6.1.1). The adjacency of two operands separated by a blank indicates multiplication. This can sometimes be confusing to the user. Particularly misleading would be the case of inadvertently omitting another operator sign, resulting in a syntactically correct but logically incorrect expression. Admittedly, the juxtaposition of two

operands with no intervening operator defines multiplication in the algebraic sense. Yet, the absence of an operator symbol in a high level language is aesthetically unpleasing and could be functionally confusing.

Modifications required: a symbol used in connection with the equal sign could be used to signify assignment.

The use of a symbol (probably "*") could be introduced for multiplication in place of the blank now used. This would require a new character to indicate cross-product.

I.HALS.Defaults, Conditional Compilation and Language Restrictions

HAL/S has relatively few implementations and all are related to Shuttle software. Therefore, the distinction between basic language features and specific implementation features is not always obvious.

All of the requirements described in this section are fulfilled by HAL/S, but not all are described in the language specification. They are described in implementation oriented documents called User's Guides.

The concept of making all restrictions and limitations which are not machine-dependent a part of the language definition is basically fulfilled in HAL/S. If one considers the language definition to consist of a language specification along with an implementation-unique user's manual, then all is covered adequately.

I1.HALS.No Defaults in Program Logic
T (Satisfies)

HAL/S completely fulfills this requirement. There are no implementation-dependent defaults which affect the program logic.

I2.HALS.Object Representation Specifications Optional
T (Satisfies)

HAL/S has a large set of default capabilities for such things as data representation and procedure definition. For example, the precision of all integer and floating-point values defaults to single (Section 4.7, page 4-20; also see comments in J4). All procedures and functions default to non-reentrant (Section 3.7.2, page 3-16; Section

3.7.3, page 3-18). All default capabilities can be overridden by specific programmer action.

13.HALS.Compile Time Variables
Fails to Satisfy

HAL/S has no compile time options which are a part of the basic language definition.

Modifications required: allow the user to specify compile time variables, especially the object machine, as a part of the basic language.

14.HALS.Conditional Compilation
Partially Satisfies

HAL/S has a type of conditional compilation in the form of the "%macro" capability. This allows the referencing of implementation-dependent intentions to the language. Conceivably these macros could be changed to accommodate various object environments.

15.HALS.Simple Base Language
Partially Satisfies

HAL/S has such a base of primitive features in which the remaining set of features could be defined (Appendix G) but it is not clearly identifiable.

16.HALS.Translator Restrictions
Fails to Satisfy

HAL/S has many implementation-dependent language capabilities which are a part of the language translator itself. These are specified in an implementation unique document called a User's Guide.

17.HALS.Object Machine Restrictions
T (Satisfies)

Hardware limitations are not considered in the language translator or specification. For example, there is no built-in language feature to warn that the resources of the intended object machine are exceeded.

J.HALS.Efficient Object Representation and Machine Dependencies

As was stated in the introductory comments to Requirement I, the requirements of this group of requirements are adequately fulfilled by HAL/S, but all are not described in the language specification. All capabilities and restrictions which are unique to the translator are described in an implementation oriented User's Guide.

J1.HALS.Efficient Object Code
T (Satisfies)

HAL/S does not impose run-time costs for unneeded code. If only a basic capability is required, no unnecessary checks or "hooks" (which might be needed for more intricate applications) are provided.

J2.HALS.Optimizations Do Not Change Program Effect
T (Satisfies)

The HAL/S translators provide optimized code with no loss of reliability or correctness. (See comments on RIGID attribute under J4.)

J3.HALS.Machine Language Insertions
Partially Satisfies

Interfaces with special purpose devices are accomplished through use of the %macro (Sections 11.2.1, 11.2.2, 11.2.3). This capability could, for example, allow a supervisor call, with or without an I/O list, to a specific I/O device. The code used in all such interfaces is predetermined and is extracted from a library. HAL/S does not provide a general-purpose machine language insertion capability.

J4.HALS.Object Representation Specifications
T (Satisfies)

The HAL/S user has the capability to specify that structure data be DENSE or ALIGNED. The DENSE attribute causes all bit strings to be packed into logical machine words. This packing implies all the necessary shifting and masking required to reference the data. The ALIGNED attribute starts every data entity on a word boundary, increasing the size of the structure, but allowing more efficient code to reference it. The ALIGNED attribute is the default (Section 4.5, page 4-15).

Unless prohibited, the compiler will reorder data elements of a compool or a structure into groups of like type in order to optimize storage. The user can override this capability by specifying the RIGID attribute (Section 4.3, page 4-10, rule 3; Section 4.5, page 4-15).

J5.HALS.Open and Closed Routine Calls
Partially Satisfies

HAL/S provides an "in-line" function capability (Section 11.2.1) which eliminates parameter passing; however, all standard functions are "closed". The specification of function type is made by the programmer, and is not left up to the choice of the translator (see diagram, page 11-2). The "in-line" capability is applicable only to functions and not procedures.

K.HALS.Program Environment

The variety of run time and compile time support packages associated with HAL/S contribute to making it powerful as a high level programming system. HAL/S is designed to be used for implementation of visible and reliable, yet complex, embedded computer systems.

K1.HALS.Operating System Not Required
T (Satisfies)

HAL/S has a stand-alone run capability. The primary operation mode is with an operating system which is directly related to the object machine.

K2.HALS.Program Assembly
T (Satisfies)

HAL/S modules are separately defined and separately compilable. Programs and compools are always compiled alone. Functions and procedures may be internal to a program (or function, or procedure) or they may be compiled separately (external feature) (Section 3.1).

The problems of type checking, efficient object representation and constant expression evaluation are shown by use of the "template" approach to building complex programs. If a HAL/S compilation contains invocations of external programs, procedures, or functions or references to variables in compools, the block template for those external blocks must appear in the compilation. Depending on the implementation, templates are generated automatically by the compiler for each

compilation unit as it is being compiled (Section 3.6). These templates are placed in a library where they may be combined with other compilation units to form complex programs (see comments under F4 and F6).

K3.HALS.Software Development Tools
T (Satisfies)

Associated with HAL/S is a powerful set of programming aids which are not a part of the language or the compiler. Below are listed a few of these support packages:

1. Simulation Data Files - The simulation data file for each compilation unit is a table of symbolic data used as a source of post-compilation statistics and as a run-time debugging aid.

2. HALLINK - The HALLINK program is used in System/360 applications to replace the standard link edit step. It determines the space requirements and creates the stack for each program or task. In performing this function, HALLINK makes use of the standard OS/360 linkage editor.

3. HALSTAT - The HALSTAT package is executed following the link/build step and prior to actual execution. It describes interrelationships between elements of a program complex and provides static verification.

K4.HALS.Translator Options
T (Satisfies)

HAL/S has a variety of compile time options which aid the user in debugging, testing, and documenting his program. The options include all of those mentioned, plus others. Not all are user controlled (e.g., indentation of lines to denote nesting and to aid documentation is the standard output mode).

K5.HALS.Assertions and Other Optional Specifications
Partially Satisfies

The optional features of assertions, debugging statements, definitions of units of measure, etc. are not a part of HAL/S. However, there exists in HAL/S a mechanism to insert statements of this type and have them treated as comments until such capabilities are developed. An

assertion statement could be denoted by a special letter (say "A") in column one of the input line and alternatively treated as a comment or a post-compile input statement, dependng on a compile-time option.

L.HALS.Translators

The requirements of this section are for the most part implementation-dependent and therefore somewhat difficult to assess with respect to the HAL/S language.

The requirement that a translator must be written in the subject language seems somewhat arbitrary. HAL/S is sufficiently flexible that such a translator could be built but thus far none has been. However, the tone of this requirement is that it "must have been done" rather than "the capability should exist to do so".

L1.HALS.No Superset Implementations
T (Satisfies)

HAL/S has no source language features which are not defined in the language specification.

L2.HALS.No Subset Implementations
T (Satisfies)

The HAL/S implementations employ the total set of language features; there are no subsets of the language.

L3.HALS.Low-Cost Translation
Unknown

Comparison data needed to determine the relative efficiency of the HAL/S compilers is not available. There is nothing inherent in the language design that precludes efficient compilers.

L4.HALS.Many Object Machines
Partially Satisfies

HAL/S is implemented for several object machines. Nothing would preclude its being adapted to others.

L5.HALS.Self-Hosting Not Required
T (Satisfies)

HAL/S compilers are not necessarily written to execute on the object machine. The shuttle implementation employs two compilers, both run on the IBM-360. One produces 360 object code, the other AP-101 (shuttle on-board computer) code.

L6.HALS.Translator Checking Required
T (Satisfies)

HAL/S compilers perform complete syntax and type checking and do no error correction.

L7.HALS.Diagnostic Messages
Fails to Satisfy

HAL/S has a full set of compile time diagnostic messages, complete with severity levels. However, these messages are not given as a part of the language definition.

Modifications required: it would be simple to choose a basic set of diagnostic messages and make them a part of the language definition.

L8.HALS.Translator Internal Structure
T (Satisfies)

Compiler capabilities in HAL/S have thus far been determined by needs and restrictions of the various implementations.

L9.HALS.Self-Implementable Language
Fails to Satisfy

HAL/S has relatively few implementations and none have compilers written in the HAL/S language. However, there is nothing in the HAL/S language to prevent self-implementation.

Modifications required: write a self-compilng HAL/S translator.

M.HALS.Language Definition, Standards, and Control

The HAL/S language is well defined. It can be learned and understood from the available documentation - Language Specification, User's Manual, and Programmer's Guide. Current implementations of HAL/S do not employ the language control features required in this section.

M1.HALS.Existing Language Features Only
T (Satisfies)

HAL/S is a modern programming language employing features considered to be within the state-of-the-art. All modifications needed to bring HAL/S up to the TINMAN requirements are feasible using known techniques and language design methods.

M2.HALS.Unambiguous Definition
T (Satisfies)

The language specification for HAL/S is concisely written and easily understood. The syntax diagrams for each feature are accompanied by sets of general and specific semantic rules. Examples of use are provided in the more extensive Programmer's Guide. The language is described in BNF notation.

M3.HALS.Language Documentation Required
T (Satisfies)

The reference documents for HAL/S are complete and easily understood. (See comments under M2.) (Reference Appendix G)

M4.HALS.Control Agent Required
T (Satisfies)

HAL/S is under control of NASA-JSC for development and maintenance.

M5. HALS.Support Agent Required
T (Satisfies)

HAL/S is under control of NASA-JSC for development and maintenance.

M6.HALS.Library Standards and Support Required
T (Satisfies)

HAL/S is under control of NASA-JSC for development and maintenance.

Section Vc - HALS Language Features Not Needed

#.HALS.Features Not Needed To Support TINMAN

There are many features built into the HAL/S language which are not required in TINMAN. Some of these features should be removed in order to comply with a specific requirement, and some could be removed without destroying the versatility of the language. However, some of the features not required would badly damage the language capabilities if removed.

Below are listed the features which should be removed in order to be compatible with requirements.

1. Implicit conversion in assignment statements (7.3)

2. Use of the "[cent]" sign for introduction of the "escape mechanism" in character literals (2.3.3)

Below are listed the features which could be removed without detriment to the language.

1. Multiline input format (2.4)

2. Use of "C" in column one of input line to denote a comment (2.5)

The features listed below are not specifically required but in many instances help HAL/S to fulfill the intent of a requirement. All of these features greatly add to the power of the language and if removed would severely limit its versatility.

1. REPLACE macro (4.2)

2. Matrix and vector-data types and built-in operations (4.7, 6.1.1)

3. SUBBIT operation (6.5.4)

4. %macro (11.2.2)

5. TEMPORARY variables (for other than loop control) (11.3)

6. EQUATE facility (11.5)

7. Use of radix in explicit conversion (6.5.2, 6.5.3)

8. SEND ERROR (9.2)

9. Full typing on pointer variables (11.4.1)

10. CAT operator for bit strings (6.1.2).

//.HALS.Discussion and Summary

Considering the total set of TINMAN requirements, HAL/S does not meet DoD specifications for a common language. However, the language could be expanded to provide some of the capabilities required and thus make HAL/S a base for the design of a common language. The following table is a breakdown of HAL/S compliance with the requirements:

REQUIREMENT			HAL/S
A.	Data and Types		
	A1.	Typed Language	T
	A2.	Data Types	P
	A3.	Precision	T
	A4.	Fixed-Point Numbers	F
	A5.	Character Data	F
	A6.	Arrays	T
	A7.	Records	T
B.	Operations		
	B1.	Assignment and Reference	T
	B2.	Equivalence	T
	B3.	Relationals	T
	B4.	Arithmetic Operations	P
	B5.	Truncation and Rounding	P
	B6.	Boolean Operations	T
	B7.	Scalar Operations - On arrays	T
	B8.	Type Conversion - Implicit	F
	B9.	Type Conversion - Explicit	P
	B10.	I/O Operations	P
	B11.	Power Set Operations	F
C.	Expressions and Parameters		
	C1.	Side Effects	T
	C2.	Operand Structure	T
	C3.	Expressions Permitted	T
	C4.	Constant Expressions	T
	C5.	Consistent Parameter Rules	T
	C6.	Type Agreement in Parameters	T
	C7.	Formal Parameter Kinds	T
	C8.	Formal Parameter Specification	F
	C9.	Variable Numbers of Parameters	F
D.	Variables, Literals and Constant		
	D1.	Constant Value Identifiers	T
	D2.	Numeric Literals	U
	D3.	Initial Values of Variables	T

D4.	Numeric Range and Step Size	F
D5.	Variable Types	P
D6.	Pointer Variables	T

E. Definition Facilities

E1.	User Definitions Possible	P
E2.	Consistent Use of Types	F
E3.	No Default Declarations	F
E4.	Can Extend Existing Operators	F
E5.	Type Definitions	F
E6.	Data Defining Mechanisms	F
E7.	No Free Union or Subset Types	T
E8.	Type Initialization	F

F. Scope and Libraries

F1.	Separate Allocation and Access Allowed	T
F2.	Limiting Access Scope	T
F3.	Compile Time Scope Determination	T
F4.	Libraries Available	T
F5.	Library Contents	T
F6.	Libraries and Compools Indistinguishable	T
F7.	Standard Library Definitions	T

G. Control Structures

G1.	Kinds of Control Structures	P
G2.	The GOTO	T
G3.	Conditional Control	P
G4.	Iterative Control	P
G5.	Routines	F
G6.	Parallel Processing	T
G7.	Exception Handling	T
G8.	Synchronization and Real-Time	T

H. Syntax and Comment Conventions

H1.	General Characteristics	T
H2.	No Syntax Extensions	T
H3.	Source Character Set	T
H4.	Identifiers and Literals	T
H5.	Lexical Units and Lines	P
H6.	Key Words	T
H7.	Comment Conventions	P
H8.	Unmatched Parentheses	T
H9.	Uniform Referent Notation	T
H10.	Consistency of Meaning	F

I. Default, Conditional Compilation and Language Restrictions

I1.	No Defaults in Program Logic	T
I2.	Object Representation Specifications Optional	T
I3.	Compile Time Variables	F
I4.	Conditional Compilation	P

	I5.	Simple Base Language	P
	I6.	Translator Restrictions	F
	I7.	Object Machine Restrictions	T
J.	Efficient Object Representations and Machine Dependencies		
	J1.	Efficient Object Code	T
	J2.	Optimizations Do Not Change Program Effect	T
	J3.	Machine Language Insertions	P
	J4.	Object Representation Specification	T
	J5.	Open and Closed Routine Calls	P
K.	Program Environment		
	K1.	Operating System Not Required	T
	K2.	Program Assembly	T
	K3.	Software Development Tools	T
	K4.	Translator Options	T
	K5.	Assertions and Other Optional Specifications	P
L.	Translators		
	L1.	No Superset Implementations	T
	L2.	No Subset Implementations	T
	L3.	Low-Cost Translation	U
	L4.	Many Object Machines	P
	L5.	Self-Hosting Not Required	T
	L6.	Translator Checking Required	T
	L7.	Diagnostic Messages	F
	L8.	Translator Internal Structure	T
	L9.	Self-Implementable Language	F
M.	Language Definition, Standards and Control		
	M1.	Existing Language Features Only	T
	M2.	Unambiguous Definition	T
	M3.	Language Documentation Required	T
	M4.	Control Agent Required	T
	M5.	Support Agent Required	T
	M6.	Library Standards and Support Required	T
Totals	T	(Fully Satisfies)	59
	P	(Partially Satisfies)	18
	F	(Fails to Satisfy)	19
	U	(Unknown)	2

Many of the features required in the TINMAN have not been found necessary for embedded spacecraft computer systems. These include extensibility, generic procedures, and recursion. If these requirements were removed, HAL/S would only fail in the following ten areas:

1. Fixed-point numbers (A4, D4)

2. Character sets as enumeration types (A5)
3. Implicit conversions (B8)
4. Power set operations (B11)
5. Default declarations (E3)
6. Consistency of meaning (H10)
7. Compile time variables (I3)
8. Translator restrictions (I6)
9. Diagnostic messages (L7)
10. Self-implementable language (L9).

As possible modifications to the language, none of these are insurmountable problems.

HAL/S, considered as a base language for the DoD HOL:

a. Is compliant with the majority of existing TINMAN requirements

b. Could be easily changed to meet approximately 13 of the requirements it does not fully meet

c. Could be changed with moderate difficulty in approximately 12 of the requirements it does not fully meet

d. Would be very difficult, but not impossible to change in approximately 14 of the remaining requirements it does not fully meet

e. Is a viable base technically, but is probably a poorer choice than some of the larger, more widely used languages.

Section VI - COMMENTS ON TINMAN REQUIREMENTS

Section VIa - TINMAN General Comments

1. Many of the TINMAN requirements are arbitrary and matters of personal taste. To a large extent, this is unavoidable since good language design and definition is an art and even the experts do not agree on what "good" is.

   The following are concepts on which there is legitimate disagreement:

   a. Defaults

   b. Implicit conversions from one data type to another

   c. Designation by the user of open or closed subroutines.

2. There are various contradictions in the TINMAN requirements. Some examples are:

   a. Within a specific requirement, consider B2 and its comments on comparison "(regardless of type)" in the header and "restricted to data of the same type" in the commentary.

   b. Across requirements, (G4) wants a special structure, namely, WHILE DO; another requirement (H1) says that there should be no special cases.

   c. Across requirements, (I5) specifies a simple base language, but one which satisfies all other requirements. The 98 requirements for TINMAN cannot be fulfilled by a simple base language.

3. Much of the text of the requirement is a justification rather than a clarification. As examples in (A3) the sentence beginning "Machine Independence", in (A4) the first sentence, and in (B11) the first sentence.

4. Some languages will satisfy - or fail to satisfy - some

requirements by cascading or by default, i.e., the requirements are not independent. This could make the counting of Fully, Partially and Fails for each language somewhat misleading.

As examples,

a. E2, 4, 5, 6, 8 are failed by COBOL by default.

b. H2 and H9 are satisfied by COBOL by default.

c. Failure of a language with respct to E1 will automatically mean failure with respect to E4.

5. Some requirements are ambiguous. For example, "fixed-point" in A4 and D4, and "equivalence/identity" in B2 are ambiguous.

6. Many of the requirements pertain to the implementers and not to the language specification. Certainly it is appropriate for TINMAN (and its successor lists of requirements) to impose requirements on the implementers, but they should be separated from the language requirements. Furthermore, many of these implementation requirements are vague and/or not measurable. For example; wanting good object code is obviously desirable, but there is no concrete way within the scope of this study to evaluate a language with respect to that requirement. As another example; it is certainly desirable that numeric data should be converted the same way at compile time and object time, but that has nothing to do with language specifications.

7. An almost fatal effect in the TINMAN requirements is the failure to provide a ranking of requirements, in terms of their importance to the intended area of application. To the extent that tradeoffs are required in the language design, it should be made clear which requirements are more important. In doing this, the translator and/or maintenance requirements should be separated and ranked separately.

8. There is a significant emphasis on libraries. While this is desirable, it seems to show a lack of faith in the viability, effectiveness, and usefulness of the eventual language. Obviously, no language is self-sufficient enough to make libraries unnecessary; however, if the language is really going to be good, TINMAN appears to be over-emphasizing this need.

A.TINMAN.Data and Types

The fixed-point requirement seems to force an almost redundant capability in implementations which also have integer and real facilities. It would seem that the HAL/S approach of making them mutually exclusive based on hardware requirements would be an acceptable compromise.

A4.TINMAN.Fixed-Point Numbers

This requirement is ambiguous. In general terminology, fixed-point numbers are sometimes interpreted as numbers between plus and minus 1; in other cases, fixed-point numbers are of the form NNN.NNN with any number of digits on either side of the decimal point.

B1.TINMAN.Assignment and Reference

The requirements say "The user will be able to declare variables for all data types". This seems to be a comment which has nothing to do with the basic requirement of B1 and which is already dealt with under requirement A2.

B9.TINMAN.Changes in Numeric Representation

The requirement that: "There will be a run time exception condition when any integer or fixed-point value is truncated" is ambiguous. It is not clear whether the requirement means that there should be an automatic transfer to some predetermined routine whenever this error occurs, or whether the programmer can specify what is to happen if this error occurs. COBOL, for example, provides the "ON SIZE ERROR" clause in the five arithmetic verbs which permits the programmer to determine what is to happen if truncation or other errors (such as division by zero) occur.

C.TINMAN.Expressions and Parameters

The "generic procedure" capability of TINMAN seems to violate the purpose of having a strongly typed language with multiple compiler checks for type mismatches (A1, C6). It would cause the compiler to be cumbersome and would be very costly to the user at compile time and run time.

D1.TINMAN.Constant Value Identifiers

The use of identifiers to represent constant values is desirable and present in most HOLs. However, the implication that section and paragraph names have constants assigned to them does not make sense within the overall spirit of TINMAN's avoidance of tricky coding.

D2.TINMAN.Numeric Literals

The requirement that "numeric constants will have the same value...in both programs and data..." is a directive to the implementer and not a language specification.

E.TINMAN.Definition Facilities

The TINMAN requirement for user definitions of data and operations seems to contradict the basic desire for readability and maintainability. The capability to extend a language might be useful but it would have to be very stringently controlled. This would also force very large, general compilers which could be beyond the hardware capabilities for some implementations.

G.TINMAN.Control Structures

The recursive capability (G5) seems to be another of those requirements which yield cumbersome compilers and add to execution time. HAL/S does not provide recursiveness because of these penalties and the knowledge that on-board aerospace applications have little need for it. In time-critical on-board aerospace applications (the indefinite space requirements needed for recursion) cannot be afforded, nor can the possibility of an error condition due to overflow of stack sizes in real-time be allowed.

I5.TINMAN.Simple Base Language

It should be obvious that this requirement seems incompatible with the others; to satisfy all the other TINMAN requirements will require a powerful - and not a simple - language.

J.TINMAN.Efficient Object Representation and Machine Dependencies

The general tone of the requirements of I and J are more translator

oriented than general language definition oriented. This requires looking at implementations and considering their needs and restrictions as opposed to looking at the general capabilities of a language.

The requirement to "not impose run time costs for unused generality" seems to be in direct conflict with the generic capabilities of C8, C9, and G5.

J3.TINMAN.Machine Language Insertions

There would be little difficulty in modifying most languages to satisfy the part of this requirement that involves inclusion only in compile time conditional statements if they existed (per requirement I4). However, this part of requirement J3 seems to defeat the purpose of allowing machine language instructions in the source program at all.

J5.TINMAN.Open and Closed Routine Calls

This appears to be an extremely undesirable requirement. While it is useful as long as a program is running on a specific machine with a known translator, its use can hamper efficient portability. The programmer might specify an open subroutine for a particular configuration and translator; yet when that program is moved to another machine and/or translator, it might be far more efficient to have a closed subroutine. Since the programmer has already specified which it is to be, the translator is unable to make the determination. It would be far better to leave this decision entirely in the hands of the translator.

L3.TINMAN.Low-Cost Translation

As an implementation requirement, this is in at least partial conflict with L4.